Technieken voor dynamische partiële herconfiguratie van FPGA's met een lage herconfiguratiekost

Techniques for Low-Overhead Dynamic Partial Reconfiguration of FPGAs

Brahim Al Farisi

Promotor: prof. dr. ir. D. Stroobandt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2014 - 2015

UNIVERSITEIT GENT

ISBN 978-90-8578-797-6
NUR 987
Wettelijk depot: D/2015/10.500/41

Acknowledgements

First, I would like to thank my promoter and dr. ir. Karel Bruneel for giving me the opportunity to do a PhD in the Hardware and Embedded Systems group. I am also very grateful to my promoter for the invaluable support throughout the years and when writing this dissertation.

I would also like to thank the members of my jury for the constructive feedback on this work. In this regard, I am also grateful to my colleagues Karel, Berg, Alexia and Amit for reading this dissertation and providing me with valuable feedback.

I would also like to thank my family, friends and colleagues for the support they gave me throughout my PhD.

A special thanks goes to my girlfriend Stéphanie, without whom this dissertation would probably not have come into being.

This work was made possible financially through a research scholarship of the IWT, Agency for Innovation through Science and Technology in Flanders.

Brahim Al Farisi
Ghent, 21/05/2015

Examination Commission

prof. dr. ir.	Rik Van de Walle, chairman Multimedia Lab Dept. of Electronics and Information Systems - ELIS Faculty of Engineering, Ghent University
prof. dr. ir.	Pieter Rombouts, secretary Dept. of Electronics and Information Systems - ELIS Faculty of Engineering, Ghent University
prof. dr. ir.	João M. P. Cardoso Department of Informatics Engineering - DEI Faculty of Engineering, University of Porto
prof. dr. ir.	Johan Bauwelinck Department of Information Technology - INTEC Faculty of Engineering, Ghent University
prof. dr. ir.	Stephan Wong Computer Engineering Laboratory Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology
dr. ir.	Philippe Faes Sigasi
dr. ir.	Jos Van Sas Alcatel-Lucent Bell
prof. dr. ir.	Dirk Stroobandt, advisor Dept. of Electronics and Information Systems - ELIS Faculty of Engineering, Ghent University

Samenvatting

Field-Programmable Gate Arrays (FPGA's) zijn elektronische chips die geprogrammeerd of geconfigureerd kunnen worden om eender welke digitale logische schakeling te implementeren. Enkel de grootte van de FPGA is hierbij een beperking. Een FPGA bestaat uit een matrix van programmeerbare functionele blokken die ingebed zijn in een programmeerbaar interconnectienetwerk. Er zijn verschillende types functionele blokken, waarvan de belangrijkste een logisch blok genoemd wordt. Om een FPGA te programmeren dient een binaire configuratie geladen te worden in een geheugen dat de functionaliteit bepaalt van de programmeerbare componenten. Dit geheugen wordt het *configuratiegeheugen* van de FPGA genoemd. FPGA's bieden een interessante combinatie van parallelle rekenkracht en kostprijs voor digitale producten met lagere oplages, zoals sattelieten, geavanceerde TV's en netwerkgerelateerde toepassingen. Ze bieden ook andere economische voordelen, zoals beter risicobeheer en kortere time-to-market.

Dynamische Partiële Herconfiguratie (DPH) van FPGA's laat toe om een deel van de chip te herprogrammeren zonder de uitvoering van de rest van de chip te onderbreken. Hierdoor kunnen verschillende functies, die niet tegelijk gebruikt worden, dezelfde chipoppervlakte delen en wordt de applicatie oppervlakte-efficiënter geïmplementeerd. Het gebied dat door de verschillende functies gedeeld wordt, wordt het *herconfiguratiegebied* genoemd. DPH leidt mogelijk tot het gebruik van een kleinere en dus goedkopere FPGA. Echter, vergeleken met een implementatie die geen DPH gebruikt, is er nu een relatief lange tijd nodig, de herconfiguratietijd genaamd, om de geïmplementeerde functie in het herconfiguratiegebied aan te passen. Dit gebeurt door het herschrijven van het configuratiegeheugen van de FPGA. In het algemeen, wordt de meerkost die

gepaard gaat met het herconfiguratieproces de *herconfiguratiekost* genoemd. Dit omvat ook het vermogenverbruik en de gebruikte hardwaremiddelen. Het systeem dat de herconfiguratie uitvoert wordt de *configuratiemanager* genoemd en is gewoonlijk in software geïmplementeerd. Het is verantwoordelijk voor het genereren van een nieuwe configuratie en het herprogrammeren van de FPGA. De conventionele DPH tool flow implementeert de verschillende functies in het herconfiguratiegebeid onafhankelijk van elkaar, zonder te trachten de herconfiguratiekost te reduceren.

De gecombineerde implementatietechnieken ontwikkeld in dit doctoraat reduceren de herconfiguratiekost significant vergeleken met de conventionele DPH tool flow, wanneer het aantal geïmplementeerde functies in het herconfiguratiegebied beperkt is. Het configuratiegeheugen van hedendaagse commerciële FPGA's bevat vooral configuratiebits die de toestand van het interconnectienetwerk bepalen. De technieken in dit doctoraat trachten daarom de herconfiguratiekost van de interconnecties te reduceren. Een FPGA tool flow bestaat typisch uit een opeenvolging van verschillende stappen. De gecombineerde implementatie kan plaatsvinden startend op verschillende punten in de tool flow. Initieel heb ik verschillende tool flows geëxploreerd die de gecombineerde implementatie starten op een verschillend punt in de tool flow. In deze exploratiefase werd de herconfiguratiekost uitgedrukt als het aantal bits dat in het configuratiegeheugen dient herschreven te worden tijdens de herconfiguratie. Het configuratiegeheugen van hedendaagse FPGA's is echter georganiseerd in groepen van bits, frames genaamd. Een *configuratieframe* moet volledig herschreven worden, zelfs als slecht één bit dient veranderd te worden. Daarom werden in dit doctoraat ook technieken ontwikkeld die de herconfiguratiekost verlagen in een frame-gebaseerde herconfiguratie-architectuur.

Dynamische circuitspecialisatie implementeert applicaties waarin sommige inputs, parameters genaamd, traag variëren in de tijd op een efficiënte manier. Het doet dit door off-line een configuratie te genereren die niet enkel bestaat uit 1'en en 0'en, maar ook Boolese functies van de parameters. Zo een configuratie wordt een geparametriseerde configuratie genoemd. Telkens de parameters van waarde veranderen dient de configuratiemanager enkel deze Boolese functies te evalueren en de FPGA herprogrammeren. Dy-

namische circuitspecialisatie laat erg snelle dynamische herconfiguratie toe van geparametriseerde systemen. In de Hardware en Ingebedde Systemen onderzoeksgroep is er een automatische tool flow beschikbaar, de TLUT tool flow genaamd, die automatisch parametrische configuraties kan genereren waarin enkel de configuratiebits van de logische blokken geparametriseerd zijn.

Schuifregister logische blokken of SRL's zijn speciale logische blokken waarin de configuratiebits ook georganiseerd zijn als een schuifregister waarvan de input en output toegankelijk zijn vanuit het programmeerbaar interconnectienetwerk. Deze configuratiebits zijn dus niet enkel toegankelijk gebruikmakend van de conventionele configuratiepoorten van de FPGA, maar ook vanuit de input van het schuifregister van de SRLs. SRL's kunnen daarom gebruikt worden om de configuratiebits van logische blokken te herschrijven tijdens de werking van de applicatie, vandaar de term *SRL-herconfiguratie*. Voor specifieke hand-ontworpen functies werd reeds aangetoond dat SRL's het herconfiguratieproces aanzienlijk kunnen versnellen wanneer enkel logische blokken dienen geherprogrammeerd te worden. In dit doctoraat integreerde ik SRL-herconfiguratie in de TLUT tool flow. In regelmatig gestructureerde ontwerpen wordt dezelfde module vele malen geïnstantieerd. In de uitgevoerde experimenten toonde ik dat SRL-herconfiguratie veel voordelen heeft voor regelmatig gestructureerde ontwerpen. Ik kwantificeerde de versnelling die bekomen kan worden gebruikmakend van SRL's. Daarnaast toonde ik dat, gebruikmakend van SRL's, het geheugengebruik van de configuratiemanager laag en onafhankelijk van het aantal modules kan gemaakt worden. Het geheugengebruik van de methode die gebruikt maakt van de conventionele configuratiepoort, daarentegen, is groter en groeit met het aantal modules in het ontwerp. Omdat SRL-herconfiguratie ook gebruik maakt van het programmeerbaar interconnectienetwerk, kan het mogelijk de timing van het ontwerp in het gedrang brengen. In de experimenten in dit doctoraat was deze negatieve invloed op de timing klein.

Summary

Field-Programmable Gate Arrays (FPGAs) are electronic chips that can be programmed or configured to implement any digital logic circuit, limited only by the size of the FPGA. An FPGA consists of a matrix of programmable functional blocks embedded in a programmable interconnection network. There are different types of functional blocks, of which the most important one is called a logic block. To program the FPGA, a binary configuration needs to be loaded in a memory that controls the state of the programmable components. This memory is called the *configuration memory* of the FPGA. FPGAs offer an interesting combination of parallel computing power and cost efficiency for low-volume digital products, such as satellites, high-end TVs and networking devices. They also offer various economic advantages, such as better risk management and shorter time-to-market.

Dynamic Partial Reconfiguration (DPR) of FPGAs allows to reprogram part of the chip without interfering with the execution of the rest of the chip. Using DPR, different functions can be implemented on the same chip area, therefore increasing the area efficiency. The area that is shared by the different functions, is generally called the *reconfigurable region*. DPR possibly leads to the use of smaller and thus cheaper FPGAs. However, compared to an implementation that does not use DPR, a relatively long time, called the reconfiguration time, is needed to change the function implemented in the reconfigurable region. This is done by rewriting the configuration memory of the FPGA. In general, the overhead associated with the reconfiguration process is called the *reconfiguration overhead*. This also entails the power consumption and the hardware resources used. The subsystem that performs the reconfiguration is called the *configuration manager* and is generally implemented in software. It is responsible for

reprogramming the FPGA and might be responsible for generating a new configuration. The conventional DPR tool flow implements the different functions in the reconfigurable region separately and does not optimize the reconfiguration overhead.

The combined implementation techniques presented in this dissertation significantly reduce the reconfiguration overhead compared to the conventional DPR tool flow, when the number of functions implemented in the reconfigurable region is limited. The configuration memory of current commercial FPGAs consists mostly of bits that control the state of the programmable interconnection network. The approach taken in this dissertation therefore attempts to reduce the reconfiguration overhead of the interconnections. The FPGA tool flow consists of a sequence of different steps. The combined implementation can take place starting at different points in the tool flow. I first explored several tool flows that start the combined implementation at a different point in the flow. In this first exploration, the reconfiguration overhead is expressed as the number of bits that needs to be rewritten in the configuration memory. However, the configuration memory of current commercial FPGAs is organized in groups of bits, called frames. A *configuration frame* needs to be rewritten completely, even when only one bit has to be changed. Therefore, in this dissertation also techniques are developed that reduce the reconfiguration overhead in a frame-based reconfiguration approach.

Dynamic Circuit Specialization efficiently implements designs in which a subset of input signals, called parameters, only vary infrequently. This is done by generating a configuration off-line that does not only contain constant 0's and 1's, but also bits that are Boolean functions of the parameter signals. This type of configuration is called a parametrized configuration and the Boolean functions are called parametrized bits. Whenever the parameters change value, the configuration manager only needs to evaluate the parametrized bits and reprogram the FPGA. This allows for very fast dynamic reconfiguration of parametrized systems. In the Hardware and Embedded Systems research group there is an automatic tool flow available, called the TLUT tool flow, that is able to automatically generate parametrized configurations in which only the bits of the logic blocks are parametrized.

Shift Register LUTs or SRLs are special logic blocks in which the configuration bits are also arranged as a shift register of which the input and the output are accessible from the configurable interconnection network. Therefore, these configuration bits are not only accessible through the conventional internal configuration port, but also through the shift inputs of the SRLs. SRLs can be used to rewrite the configuration bits of a logic block during run-time, hence the term *SRL reconfiguration*. For specific hand-made designs, it has previously been shown that SRLs can significantly speed-up the reconfiguration process when only logic blocks need to be reprogrammed. In this dissertation, I integrated SRL reconfiguration in the TLUT tool flow. Regularly structured designs instantiate the same hardware module many times. In the experiments, I show that for regularly structured designs, SRLs have many benefits. I quantify the speed up that can be obtained using SRLs. Furthermore, I show that, for regularly structured designs, the memory usage of the configuration manager can be kept low and independent of the number of modules. The memory usage of reconfiguration using the conventional internal configuration port, on the other hand, is greater and grows with the number of modules present in the design. Because the SRLs also make use of the programmable interconnection network, they could interfere with the timing of the design. In the experiments in this dissertation, this impact on the timing was found to be limited.

Contents

List of Figures

List of Tables

List of Acronyms

AIG	And-Inverter Graph
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CAD	Computer-Aided Design
CB	Connection Block
CLB	Configurable Logic Block
CM	Configuration Manager
DCS	Dynamic Circuit Specialization
DPH	Dynamische Partiële Herconfiguratie
DPR	Dynamic Partial Reconfiguration
EEPROM	Electrically Erasable Programmable Read-Only Memory
FF	Flip-Flop
FIFO	First In First Out
FIR	Finite Impulse Response
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port
IO	Input/Output
LUT	Lookup table
MCNC	Microelectronics Center of North Carolina
NCD	Native Circuit Description
PLA	Programmable Logic Array
PLB	Processor Local Bus
PPC	Partial Parameterized Configuration
RAM	Random Access Memory
RCF	Reduction of maximum Clock Frequency
RegExp	Regular Expression Matching

RO	Reconfiguration Overhead
RR	Reconfigurable Region
RRG	Routing Resource Graph
RRO	Reduction of the Reconfiguration Overhead
SB	Switch Block
SRAM	Static Random-Access Memory
SRL	Shift Register LUT
RegExp	Regular Expression Matching
TCAM	Ternary Content Addressable Memory
TCON	Tunable Connection
TLUT	Tunable LUT
UMTS	Universal Mobile Telecommunications System
VHDL	Very-high-speed Integrated Circuit
VTR	Verilog-To-Routing
VPR	Versatile Placement and Routing
WLI	Wire Length Increase
XDL	Xilinx Design Language

Chapter 1

Introduction

1.1 Introduction to the research

1.1.1 FPGAs and Dynamic Partial Reconfiguration

A Field-Programmable Gate Array (FPGA) is an electronic chip that can be used to implement an arbitrary digital logic circuit. It consists of an array of configurable functional blocks embedded in a configurable interconnection network. By writing the configuration memory of an FPGA with the appropriate binary values, any desired digital circuit can be implemented, limited only by the size of the FPGA.

FPGAs are, first of all, useful for implementing electronic products because they offer a different trade-off between cost and performance compared to general-purpose processors and Application-Specific Integrated Circuits (ASICs). A low cost is associated with implementations using processors, since processors are cheap and software can be developed relatively easily. However, because of their sequential nature, processors generally lack performance, especially when many parallel computations need to be performed. ASICs, on the other hand, have a very high parallel computational performance, but are associated with a very high cost, primarily due to the high non-recurrent engineering costs associated with the necessary silicon masks. Also designing an ASIC is complex and costly since physical design of the chip is necessary. That is why ASICs only make sense when the chip volumes sold are high enough. When this is not the case, FPGAs can provide a more cost-efficient alternative. Because of the flexibility of an FPGA, high chip volumes are no longer required for a single product. This is, different manufacturers

can each buy the same type of FPGA chip, but implement a completely different product by programming the chip accordingly. This allows making electronic products with a high parallel computing performance of which only a limited number is sold.

The flexibility of FPGAs also offers other economic advantages. Since the design effort for FPGAs is relatively low, FPGAs can be used in electronic products that require both a low time-to-market and a high parallel performance. From a risk management perspective, FPGAs make sense because fixing a bug is straight-forward, since it only requires reprogramming the FPGA. Note that fixing a bug after an ASIC has been manufactured is extremely expensive, since new silicon masks are required. New and upcoming products are typically first implemented using FPGAs, until the design is stable and the product volumes increase, after which using an ASIC can be considered.

Finally, FPGAs are also extensively used for testing, debugging and emulating digital chips. Because of their parallel nature they have the potential of significantly outperforming simulation in software.

Conventionally, FPGAs are used statically: they are configured once at the start-up of the chip, after which the configuration is not changed during the operation of the application. Configuring an FPGA, however, is nothing more than writing the configuration memory. Since configuration memories are typically SRAM-based this happens relatively fast - in the order of tens of milliseconds for an average FPGA chip. When only a portion of the FPGA is reconfigured, this is referred to as Partial Reconfiguration. Some FPGAs can be reconfigured partially without interrupting the functionality of the rest of the FPGA. Because with partial reconfiguration only a portion of the configuration memory is rewritten, this allows for even faster reconfiguration, in the order of a few milliseconds. When Partial Reconfiguration is performed during the run-time of the application, the term Dynamic Partial Reconfiguration (DPR) is used.

Initially, DPR was primarily used on a very large time scale, for bug fixes and system updates, since it allows for in-field updates without interrupting the system's operation. Recently, DPR is also explored on a smaller time-scale. Indeed, with DPR it is possible to implement different functions that are not needed at the same time in the system on the same FPGA area. Due to hardware sharing, the amount of necessary resources will be smaller than in the case when

each function uses a separate set of resources. The main reason to use DPR is therefore that it possibly implements an application using a smaller and thus cheaper FPGA.

The area on which the different functions are time-multiplexed is generally called the *reconfigurable region* (RR). Whenever one wants to change the implemented circuit, an amount of time, called the reconfiguration time, is needed to rewrite the configuration memory of the reconfigurable region. In general, the overhead associated with the reconfiguration process is called the *reconfiguration overhead*. This can also entail the energy consumption and the hardware resources used by the reconfiguration process. The subsystem that executes the reconfiguration process is called the *configuration manager* (CM) and is generally implemented in software. It is responsible for reprogramming the FPGA and might be responsible for generating a new configuration.

1.1.2 Dynamic Circuit Specialisation

In certain circuits a subset of input signals, called parameters, only vary infrequently. An example of such a circuit is a constant multiplier in which one of the factors only changes infrequently (and is thus considered constant between changes). In this case, the constant factor is considered to be the parameter. These parameter inputs stay the same for a relatively long period of time and hence the state of the hardware driven by these signals also does not change. Only when the parameters change value will the associated hardware driven by these signals change state. The hardware resources are therefore not used efficiently since part of the hardware is inactive most of the time.

DPR can offer a solution for this problem by moving the parameter inputs to the configuration manager (CM). The CM is in this case responsible for generating a specialised circuit with only constantly varying regular signals, optimised for values of the parameter signals. This way the inactive logic is removed. The question then remains: how do we best set up such a DPR system? A first possible solution is to run the complete FPGA tool flow on-line to generate new configurations. This takes an excessive amount of time. Another approach could be to generate all the possible circuits off-line and store them in a database. The CM then only needs to look up the right configuration and reconfigure the FPGA. This is possibly

the fastest approach. For some applications, however an enormous amount of configurations are needed. A third approach, called *Dynamic Circuit Specialization*, offers an elegant solution to this problem. This method generates a configuration off-line that does not only contain constant 0's and 1's, but also bits that are a Boolean function of the parameter signals. This type of configuration is called a parameterized configuration and the Boolean functions are called parameterized bits. Whenever the parameters change value, the CM only needs to evaluate the parameterized bits to create a specialised configuration and reconfigure the FPGA. Generating a new configuration thus boils down to evaluating Boolean functions. This can happen orders of magnitude faster than running the complete FPGA tool flow.

A tool flow that is able to automatically generate such parameterized configurations has been developed at the Hardware and Embedded Systems (HES) group at Ghent University. This tool flow is called the *TLUT tool flow* and generates parameterized configurations in which only the bits of the logic blocks are parameterized. This tool flow is available through GitHub. A more general tool flow called the Dynamic Circuit Specialization (DCS) tool flow, that also generates parameterized routing bits, is still being researched. The DCS tool flow does not map functions on a regular circuit, but on a *Tunable Circuit*. A tunable circuit is the abstraction of a dynamically reconfigurable circuit. Some prototypes of the different tools, such as a placer and router of Tunable circuits, called TPlace and TRoute, are already implemented. The complete DCS flow that can also generate parameterized routing bits, however, is not yet available.

1.1.3 Limited-context Dynamic Partial Reconfiguration

In theory, the number of circuits that can be time-multiplexed in a reconfigurable region is only limited by the size of the memory available to store the different configurations associated with the circuits. In many cases, however, only a limited number of circuits are needed. In this work we will call this special case of DPR, *limited-context DPR*. An example that uses limited-context DPR is a mobile transceiver that supports different communication standards (like CDMA or Wi-Fi), but only uses one at any given time. In this case, every context is a circuit that contains the necessary functions to support the corresponding communication standard. Another example

is an audio application which only contains a low-pass and a high-pass adaptive filter.

1.1.4 Shift-register look-up table (SRL) reconfiguration

The most common building block of an FPGA is a Lookup Table (LUT). A K-input LUT can implement any Boolean function with K inputs and one output. A K-input LUT or K-LUT consists of 2^K SRAM memory cells connected to a multiplexer. To implement a certain Boolean function, the truth table needs to be stored in the SRAM cells of the LUT.

In a *shift-register LUT* (SRL), the truth table configuration bits are also arranged as a shift register of which the input and the output are accessible from the configurable routing. Therefore these configuration bits are not only accessible through the FPGA's configuration ports, but also through the shift inputs of the SRLs. The SRL provides a very efficient means for implementing shift registers. Instead of using flip-flops of the FPGA fabric, the flip-flops of the configuration memory can be used. When only the truth table contents of LUTs need to be changed during run-time, SRLs can also be used for reconfiguring LUTs, hence the term *SRL reconfiguration*.

1.2 Focus and Contributions

1.2.1 Exploring novel tool flows for limited-context DPR

The conventional tool flow for DPR generates a configuration for each circuit by implementing it separately in the reconfigurable region. The conventional DPR flow does not optimize the reconfiguration overhead, making DPR less useful for more dynamic applications. A first example of such a dynamic application is the network intrusion detection system described in [76], where reconfiguration has to happen fast enough so that no network packets are dropped. A second example is an imaging application where the reconfiguration possibly happens per image frame [10]. In the case of [76] there are only 2 contexts, corresponding to different networking protocols. The work in [10] considers a video chain with a limited number of processing steps. These are therefore both also good examples of limited-context applications.

In this part of my dissertation, I show that the reconfiguration overhead can be reduced by combined implementation and optimization of the limited number of circuits starting at a certain point of the tool flow. The impact this has on the wire length of the implemented circuits is also discussed.

I explored three different tool flows. The first tool flow uses a technique found in literature called *edge matching*. When I started my research, edge matching was the only technique I found in literature that also considered a combined implementation approach to reduce the reconfiguration overhead. Edge matching tries to place the LUTs of the different circuits in such a way that the number of connections that have the same source and sink is maximized. These overlapping connections do not need to be reconfigured. Edge matching attempts to reduce the reconfiguration overhead by reducing the number of connections that need reconfiguration. This first tool flow reuses the placement and routing tools of the DCS tool flow, called TPlace and TRoute. This work was presented in the 'Design, Automation, and Test in Europe Conference and Exhibition' in 2013.

> **Brahim Al Farisi**, Karel Bruneel, João M. P. Cardoso and Dirk Stroobandt,
> "An automatic tool flow for the combined implementation of multi-mode circuits",
> *Proceedings 2013 Design, Automation, and Test in Europe Conference and Exhibition (DATE 2013)*,
> pp. 821-826

The second tool flow optimizes the *total wire length* of a Tunable circuit that is formed during a combined placement step. In the third tool flow the *individual wire lengths* of the different circuits are optimized during placement and TRoute is used to perform the routing. These last two tool flows are presented in the 'IEEE Computer Society Annual Symposium on VLSI' in 2013.

> **Brahim Al Farisi**, Elias Vansteenkiste, Karel Bruneel and Dirk Stroobandt,
> "A novel tool flow for increased routing configuration similarity in multi-mode circuits",
> *Proceedings 2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2013)*,
> pp. 96-101

1.2.2 Reducing the overhead for limited-context DPR in a frame-based reconfiguration approach

In the first exploration of tool flows, discussed in the previous section, the reconfiguration overhead is expressed as the number of bits that needs to be rewritten in the configuration memory. However, the configuration memory of current commercial FPGAs is organized in frames. A configuration frame needs to be rewritten completely, even when only one bit needs to be changed. Although much lessons were learned from the techniques developed in the first exploration of tool flows, these would only be useful if the routing's configuration memory would be bit-addressable. In this part of the dissertation other techniques are developed that can be used in a frame-based reconfiguration approach. The main aim in this part of the dissertation is to cluster dynamic bits in fewer configuration frames. These techniques were developed based on the lessons learned in the first exploration of tool flows.

I first developed a frame-based router called StaticRoute, in which routing happens in two steps. In a first step the configuration memory of the reconfigurable region's routing switches is divided into a static and a dynamic part. Care needs to be taken that the memory cells of the static part reside in other frames than those of the dynamic part. Then, in a second step the interconnections of *all* circuits are routed simultaneously using StaticRoute in such a way that dynamic bits are avoided in the static part. The dynamic bits are thus clustered in the dynamic part of the configuration memory. To the best of my knowledge, I was the first to propose such an approach. The StaticRoute tool flow was first presented in the 'International Conference on Field Programmable Logic and Applications' in 2013.

> **Brahim Al Farisi**, Karel Bruneel and Dirk Stroobandt,
> "StaticRoute : a novel router for the dynamic partial reconfiguration of FPGAs",
> *Proceedings 2013 International Conference on Field Programmable Logic and Applications (FPL 2013)*,
> pp. 1-7

An overview of the work done in the previous conference papers regarding limited-context DPR is given in a paper that was presented at the PhD forum of the 'International Conference on Field Programmable Technology' in 2014.

Brahim Al Farisi, Karel Heyse and Dirk Stroobandt,
"Reducing the overhead of dynamic partial reconfiguration for multi-mode circuits",
Proceedings 2014 International Conference on Field-Programmable Technology (ICFPT 2014),
pp. 1-2

The experiments in the FPL2013 paper regarding the StaticRoute tool flow only implemented 2 circuits in the reconfigurable region. They were done on a simple 4-LUT based architecture and only looked at the total wire length of a circuit as a metric for performance. Also, only the switch blocks were considered for the static part. These experiments were extended for publication in the 'Design Automation for Embedded Systems Journal'. In contrast to the FPL2013 paper, in this journal paper it is explored how the static part is best chosen. In this more thorough exploration also the connection blocks are considered. Furthermore, experiments that implement more than 2 circuits in the reconfigurable region are presented. The experiments are also done on a realistic FPGA architecture with 6-LUTs and more complex configurable logic blocks, based on the commercial Altera Stratix IV FPGA. Finally, the actual impact on the maximum attainable clock frequency is assessed using the timing analyser available in the academic VTR framework.

Brahim Al Farisi, Karel Heyse, Karel Bruneel, João Cardoso and Dirk Stroobandt,
"Enabling FPGA routing configuration sharing in dynamic partial reconfiguration",
Design Automation for Embedded Systems,
to be published (Accepted 2014)

The StaticRoute tool flow requires an extra step in which certain configuration frames are manually marked as static. This means that these configuration frames should not contain dynamic bits in the end result. The ClusterRoute tool flow is similar. The main difference is that the router works completely automatic and it only needs to know how the routing configuration bits are organized in frames. A journal publication regarding the ClusterRoute tool flow has been submitted to the 'ACM Transactions on Design Automation of Electronic Systems' journal.

Brahim Al Farisi, Karel Heyse, João Cardoso and Dirk Stroobandt,
"ClusterRoute: a novel router for low-overhead dynamic partial reconfiguration of a limited number of circuits",
ACM Transactions on Design Automation of Electronic Systems,
submitted

1.2.3 Integration of SRL reconfiguration in the TLUT tool flow

The second part of my dissertation covers the research done about shift register LUT (SRL) reconfiguration. Before my research started, SRLs had been primarily used to reconfigure specific hand-designed functions. In this part of my dissertation, I explain how I integrated SRL reconfiguration in the automatic TLUT tool flow. This required adding appropriate timing constraints to make the tools treat the SRLs between reconfigurations as conventional LUTs. Also the design of a hardware block, called HWSRL, is discussed that interfaces between the configuration manager and the reconfigurable hardware. Finally, in order to make the truth table bits of multiple LUTs accessible from the configuration manager, they are grouped and each group is arranged as a larger shift register, called a reconfiguration path, by connecting the shift out of an SRL to the shift in of the next SRL. This work is an extension of my Master Thesis and was presented at the 'ACM/SIGDA International Symposium on Field programmable Gate Arrays' in 2010.

Brahim Al Farisi, Karel Bruneel, Harald Devos and Dirk Stroobandt,
"Automatic tool flow for shift-register-LUT reconfiguration: making run-time reconfiguration fast and easy",
Proceedings 2010 ACM/SIGDA International Symposium on Field programmable Gate Arrays (FPGA 2010),
pp. 287-287

1.2.4 Exploring the benefits and overhead of SRL reconfiguration for regularly structured designs

In regularly structured designs the same hardware module is instantiated many times. An example of a regularly structured design is an adaptive filter in which a block that performs a multiply-accumulate

operation is instantiated many times. For regularly structured designs, I quantify the speed-up of the reconfiguration process when using the new automatic tool flow using SRL reconfiguration, compared to reconfiguration using a conventional internal FPGA configuration port. I also show that the memory efficiency thereof can be greatly improved using SRLs. The impact of SRL reconfiguration on the maximum operating clock frequency of the design is also discussed. This work was presented in the 'International Conference on Field Programmable Logic and Applications' in 2011.

> **Brahim Al Farisi**, Karel Heyse, Karel Bruneel and Dirk Stroobandt,
> "Memory-efficient and fast run-time reconfiguration of regularly structured designs",
> *Proceedings 2011 International Conference on Field Programmable Logic and Applications (FPL 2011)*,
> pp.1-6

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows. The next two chapters are background chapters. Chapter 2 gives an overview of the architecture of FPGAs and the tool flow needed to generate a configuration that implements a desired functionality. It also discusses the different ways FPGAs can be programmed or configured, including Shift-Register LUT (SRL) reconfiguration. Chapter 3 focuses on Dynamic Partial Reconfiguration (DPR) and more specifically Dynamic Circuit Specialisation and limited-context DPR. This chapter also gives an overview of the TLUT tool flow.

The research presented in this dissertation consists of two main parts. The first part describes the work that aims at lowering the reconfiguration overhead of limited-context DPR. This parts is described in Chapter 4 and Chapter 5. The second part presents the work done regaring SRL reconfiguration, it includes Chapter 6 and Chapter 7.

In Chapter 4, a first exploration of different tool flows for limited-context Dynamic Partial Reconfiguration is presented. Based on the lessons learned in Chapter 4, two tool flows were developed for low-overhead limited-context DPR: the StaticRoute tool flow and the ClusterRoute tool flow. These tool flows are presented in Chapter 5.

As explained earlier, the StaticRoute tool flow requires marking some configuration frames as static. StaticRoute routes the connections of all circuits in such a way that the dynamic bits are clustered in the dynamic frames. This is discussed in Section 5.1. The ClusterRoute tool flow, on the other hand, automatically clusters the dynamic bits in fewer configuration frames. An extra step in which some frames are marked as static is not longer needed. This is discussed in detail in Section 5.2.

The second part of this dissertation starts with an introduction to SRL reconfiguration in Chapter 6. Chapter 7 presents how SRL reconfiguration was integrated in the TLUT tool flow. It also explains how SRL reconfiguration can be beneficial for regularly structured designs.

Both parts of this dissertation are concluded in Chapter 8. Possible future research is also presented there.

1.4 Publications

Journal papers

Karel Heyse, **Brahim Al Farisi**, Karel Bruneel and Dirk Stroobandt, "TCONMAP: Technology Mapping for Parameterised FPGA Configurations", *ACM Transactions on Design Automation of Electronic Systems*, to be published (Accepted 2015)

Brahim Al Farisi, Karel Heyse, João Cardoso and Dirk Stroobandt, "ClusterRoute: a novel router for low-overhead dynamic partial reconfiguration of a limited number of circuits", *ACM Transactions on Design Automation of Electronic Systems*, under review

Karel Heyse, Jente Basteleus, **Brahim Al Farisi**, Dirk Stroobandt, Oliver Kadlcek and Oliver Pell, "On the impact of replacing low-speed configuration buses on FPGAs with the chip's internal configuration infrastructure", *ACM Transactions on Reconfigurable Technology and Systems*, to be published (Accepted 2014)

Brahim Al Farisi, Karel Heyse, Karel Bruneel, João Cardoso and Dirk Stroobandt, "Enabling FPGA routing configuration sharing in dynamic partial reconfiguration", *Design Automation for Embedded Systems*, to be published (Accepted 2014)

Elias Vansteenkiste, **Brahim Al Farisi**, Karel Bruneel and Dirk Stroobandt, "TPaR : place and route tools for the dynamic reconfiguration of the FPGA's interconnect network", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2014)

Conference papers with international peer review

Brahim Al Farisi, Karel Heyse and Dirk Stroobandt, "Reducing the overhead of dynamic partial reconfiguration for multi-mode circuits", *Proceedings 2014 International Conference on Field-Programmable Technology (ICFPT 2014)*, pp. 1-2

Brahim Al Farisi, Elias Vansteenkiste, Karel Bruneel and Dirk Stroobandt, "A novel tool flow for increased routing configuration similarity in multi-mode circuits", *Proceedings 2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2013)*, pp. 96-101

Brahim Al Farisi, Karel Bruneel and Dirk Stroobandt, "StaticRoute : a novel router for the dynamic partial reconfiguration of FPGAs", *Proceedings 2013 International Conference on Field Programmable Logic and Applications (FPL 2013)*, pp. 1-7

Brahim Al Farisi, Karel Bruneel, João M. P. Cardoso and Dirk Stroobandt, "An automatic tool flow for the combined implementation of multi-mode circuits", *Proceedings 2013 Design, Automation, and Test in Europe Conference and Exhibition (DATE 2013)*, pp. 821-826

Karel Heyse, **Brahim Al Farisi**, Karel Bruneel and Dirk Stroobandt, "Automating reconfiguration chain generation for SRL-based run-time reconfiguration", *Lecture Notes in Computer Science: Reconfigurable Computing: Architectures, Tools and Applications (2012)*, pp.1-12

Brahim Al Farisi, Karel Heyse, Karel Bruneel and Dirk Stroobandt, "Memory-efficient and fast run-time reconfiguration of regularly structured designs", *Proceedings 2011 International Conference on Field Programmable Logic and Applications (FPL 2011)*, pp.1-6

Robbe Vancayseele, **Brahim Al Farisi**, Wim Heirman, Karel Bruneel and Dirk Stroobandt, "RecoNoC : a reconfigurable network-on-chip", *Proceedings 2011 International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2011)*, pp. 1-2

Brahim Al Farisi, Karel Bruneel, Harald Devos and Dirk Stroobandt, "Automatic tool flow for shift-register-LUT reconfiguration: making run-time reconfiguration fast and easy (abstract only)", *Proceedings 2010 ACM/SIGDA International Symposium on Field programmable Gate Arrays (FPGA 2010)*, pp. 287-287

Chapter 2

FPGAs: Architecture, Tools and Configuration

In the first section of this chapter the basic architecture of FPGAs is presented. The subsequent section describes the typical FPGA tool flow that implements a design on the FPGA. The FPGA tool flow generates a configuration that can be loaded into the FPGA's configuration memory. Finally, the different ways an FPGA can be programmed or configured are presented.

2.1 FPGA Architecture

An FPGA basically consists of an array of programmable or configurable functional blocks that are embedded in a configurable interconnection network. Modern FPGAs also contain hard-wired embedded processors.

2.1.1 Configurable functional blocks

The most common functional block is called a Configurable Logic Block (CLB). An example of a simple CLB is shown in Figure 2.1. This CLB contains one Lookup Table (LUT) with two inputs, also called a 2-LUT, and one flip-flop. A K-input LUT or K-LUT consists of 2^K SRAM memory cells connected to a multiplexer. A K-input LUT can implement any Boolean function with K inputs and one output. To implement a certain Boolean function, the truth table needs to be generated and stored in the SRAM cells of the LUT. The inputs

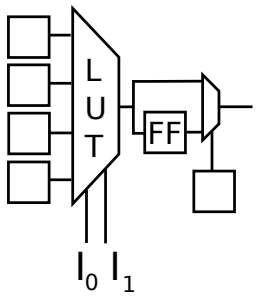


Figure 2.1: A schematic representation of a simple CLB containing one LUT and one flip-flop.

of the LUT act as an address to look up the right output value in the truth table. The FF in the CLB can be used to build sequential logic, this is logic of which the state changes synchronously with a clock. A multiplexer, also controlled by an SRAM bit, determines whether or not the output of the CLB is clocked. In modern FPGAs the FF can also be used separately, without going through the LUT, as shown in Figure 2.2. Most FPGAs used to be based on 4-LUTs. However, in more recent technology nodes a shift has been made to 6-input LUTs.

Some of the LUTs in modern FPGAs can also be used to build small localized memories. The SRAM cells of these LUTs can be used as a shift register or RAM. When the LUT is used as a shift register it is called a shift register LUT (SRL). The RAM functionality of CLBs is called distributed RAM. Figure 2.2 shows an example of a CLB of which the LUT can also be set as a distributed RAM or SRL. SRL configuration is discussed in more detail in Section 2.3.3 and Chapter 6.

In modern FPGAs the CLBs are much more complex than in Figure 2.2. CLBs of Xilinx FPGAs, for example, contain several LUTs, that are organized in groups called slices. On Figure 2.3 it can be seen that each CLB on a Virtex 5 contains two slices. Figure 2.4 shows a slice from a Virtex 5 FPGA, that contains four 6-LUTs. Between the LUTs in a CLB there are fast direct connections, as shown in Figure 2.3. The number of LUTs in a slice and number of slices in a CLB is dependent on the architecture. The CLB of a Virtex 4 FPGA for example contains four slices, each with two 4-LUTs.

As can be seen on Figure 2.4 a slice contains more than just the LUTs and FFs. Specialized hard-wired logic is also available, such as

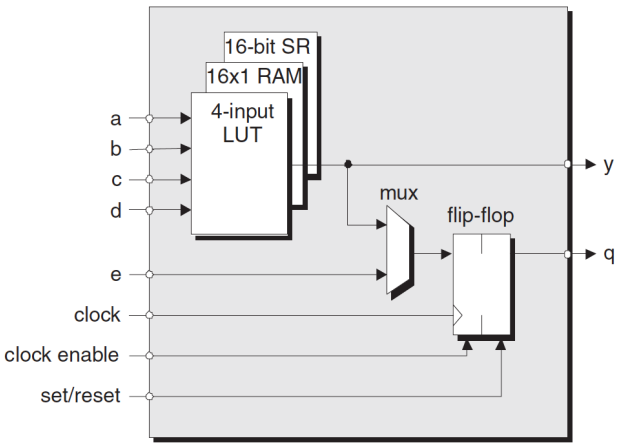


Figure 2.2: A schematic representation of a CLB in which the LUT can also be set as an SRL or DRAM.

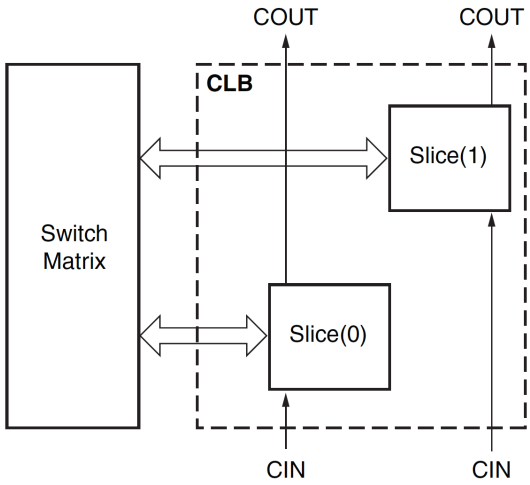


Figure 2.3: Schematic representation of the slices in a Virtex 5 CLB.

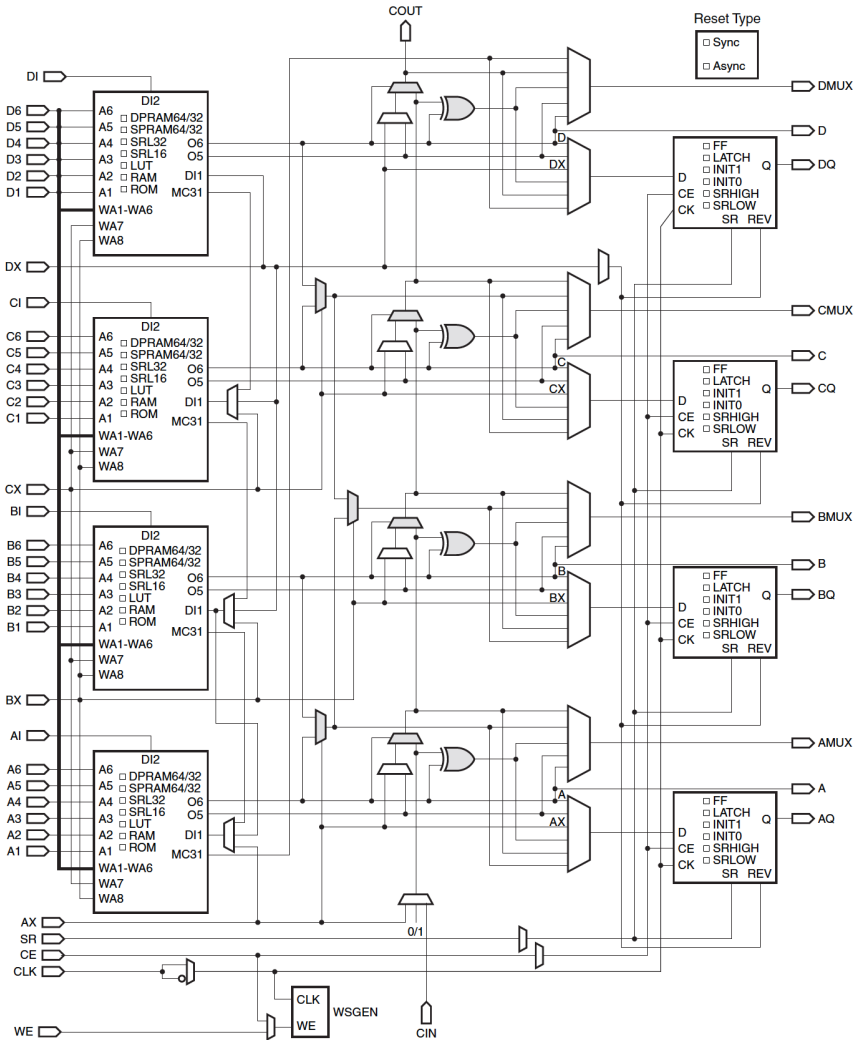


Figure 2.4: Schematic representation of a Virtex 5 slice.

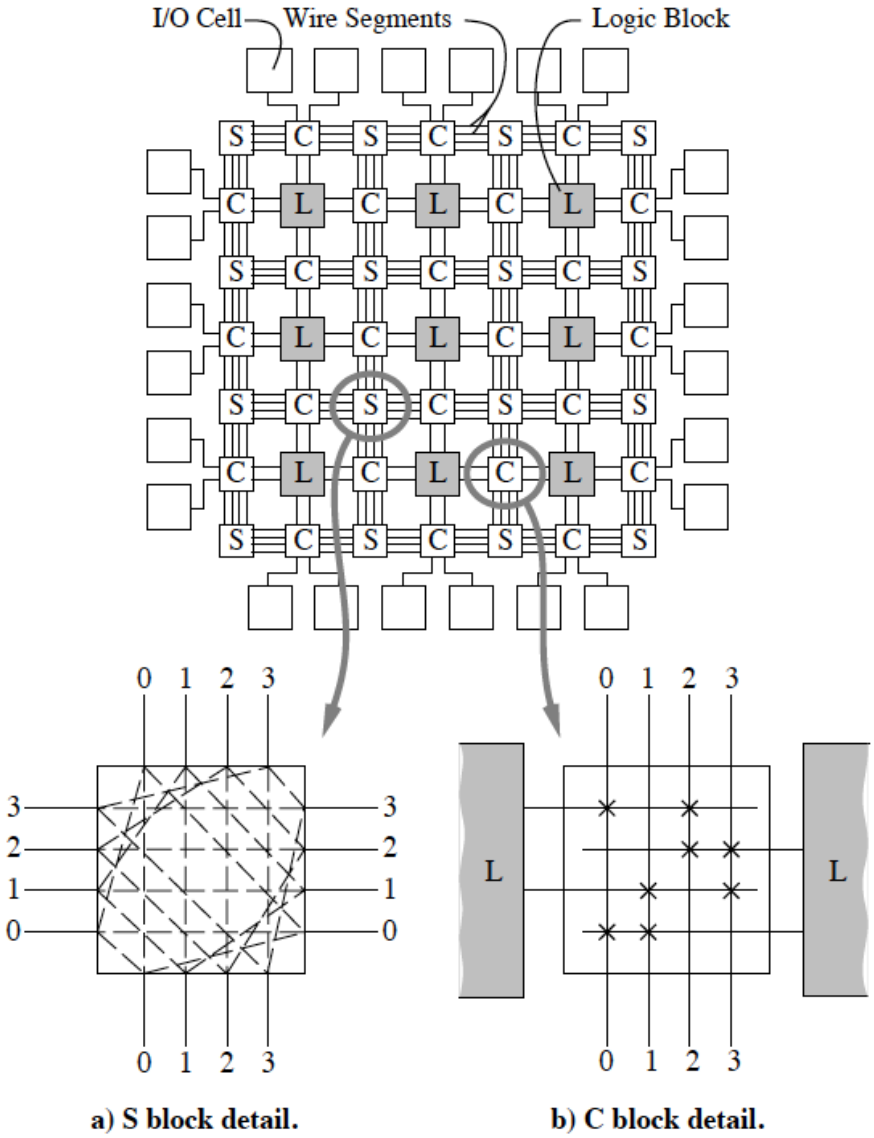


Figure 2.5: An island-style style FPGA architecture, with details on the switch and connection block components [61].

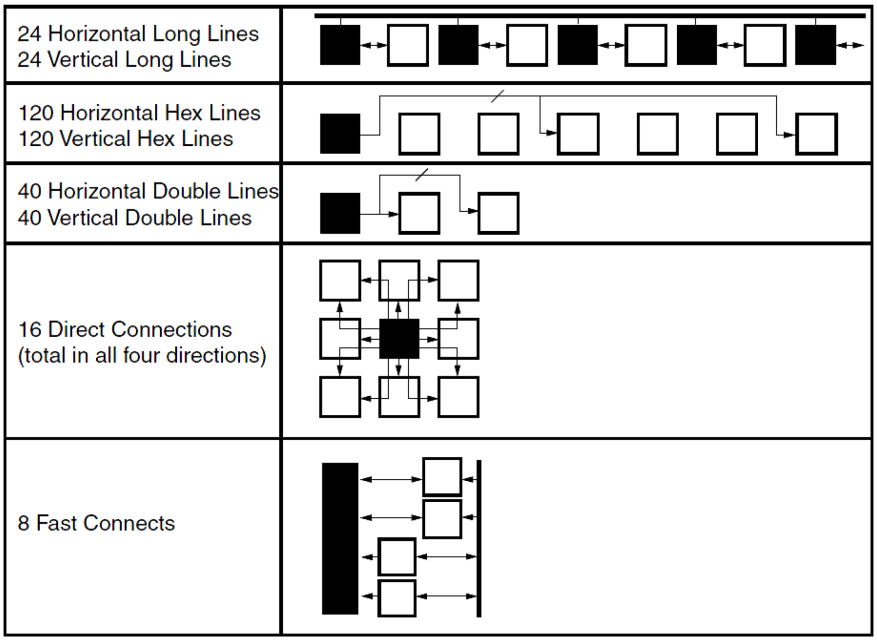


Figure 2.6: Illustration of the hierarchical build up of a programmable interconnection network.

logic gates and carry chains, that are used to build efficient adders. Because FPGAs are used a lot for signal processing applications, they also contain specialized functional blocks as hard-wired multipliers and larger RAM blocks called block RAMs or BRAMs. The CLBs, multipliers and BRAMs are organized in columns. There are also other columns available that are used for clocking purposes.

A final type of functional block is called an input-output block or IO block. These blocks are located on the edges of the FPGA and are used to connect to external hardware.

2.1.2 Configurable interconnection network

The functional blocks of an FPGA are embedded in a network of wires that can be programmed to interconnect the CLBs as needed. A group of wires between adjacent CLBs is called a channel. The number of wires in a channel is called the channel width. Figure 2.5 shows a homogeneous FPGA, with only CLBs, embedded in an interconnection network with a channel width of four wires. On the same

figure it can be seen that the programmable routing network is organized in connection blocks and switch blocks. The connection blocks can interconnect the input and output pins of a CLB with the wires in an adjacent channel. The switch blocks are used to interconnect the wires of crossing channels.

The programmable interconnection network of Modern FPGAs is based on multiplexers and uses unidirectional wires [61]. The multiplexers are built up using pass-transistors controlled by a bit value stored in an SRAM cell. The interconnection network is also mostly built up hierarchically, with many short wires and less longer wires [72]. Figure 2.6 illustrates the hierarchical build up of a programmable interconnection network.

2.1.3 Embedded processors

Modern FPGAs have evolved from purely logic matrices to complex computing structures. A first example of this are the BRAMs and hard-wired multipliers discussed above. Another important example of this evolution are embedded hard-wired processors. The embedded processor on a Virtex 5 FPGA, for example, is a PowerPC 440. Its maximum clock frequency is 500 Mhz. In recent years, Xilinx has developed the Zynq-7000 All programmable SoC, which combines a dual ARM Cortex A9 Core with an Artix-7 or Kintex-7 FPGA. The cores can run from 667 to 800 MHz [53].

If a hard-wired embedded processor is not present or a more lightweight approach is necessary, then a soft-core processor can be used, the MicroBlaze [111]. The user can scale the MicroBlaze implementation to the application's needs, because it is implemented in the logic fabric. However, it will never achieve the same performance as a hard-wired embedded processor.

2.2 Overview of the conventional FPGA Tool Flow

The SRAM cells that control the LUTs, multiplexers and programmable interconnect together make up the configuration memory. To implement a design on the FPGA, a configuration, consisting of binary values, needs to be loaded in the configuration memory. There are commercial FPGA tool flows available that can generate such configurations given a textual description of the digital design

in a Hardware Description Language (HDL). A typical FPGA flow, depicted in Figure 2.7, takes a HDL file as input and typically consists of 4 steps: synthesis, technology mapping, placement and routing.

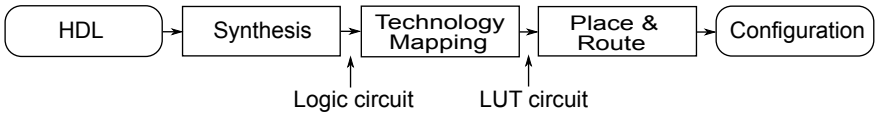


Figure 2.7: The FPGA tool flow.

2.2.1 Synthesis

The synthesis tool takes as input the HDL description and generates a technology-independent circuit with logic gates. Figure 2.8 shows a Very-high-speed Integrated Circuit Hardware Description Language (VHDL) description of a 4-to-1 multiplexer. VHDL is a standardized and commonly used HDL. We will use the multiplexer example throughout this section.

Figure 2.9 shows the synthesis result for the 4-to-1 multiplexer. This logic circuit is an and-inverter graph (AIG), a logic circuit description containing only AND and inverter gates. This format, developed at Berkeley university, can represent logic circuits very efficiently and the memory usage scales very well when used for larger circuits [16] [15] [14].In a byte-addressable memory, the two least significant bits of pointers are not used. In the textual and binary representations of AIGs of Berkeley, one of these bits is used to indicate whether or not an edge is inverted.

2.2.2 Technology mapping

The technology mapping algorithm takes in the technology-independent logic circuit and generates a technology-dependent LUT circuit, together with the Boolean functions that describe the relation between the inputs and output of each LUT. These are necessary to generate the associated truth table contents for the LUTs. The truth table of a LUT is generated using evaluation. This is done by filling in each possible input in the Boolean function and noting the result in the truth table at the position in the truth table that corresponds with the input address.

```
entity multiplexer is
port (
  I   : in    std_logic_vector (3 downto 0);
  S   : in    std_logic_vector (1 downto 0);
  O   : out   std_logic
);
end multiplexer;

architecture behavior of multiplexer is
begin
  O <= I(conv_integer(S));
end behavior;
```

Figure 2.8: VHDL code of a 4-to-1 multiplexer.

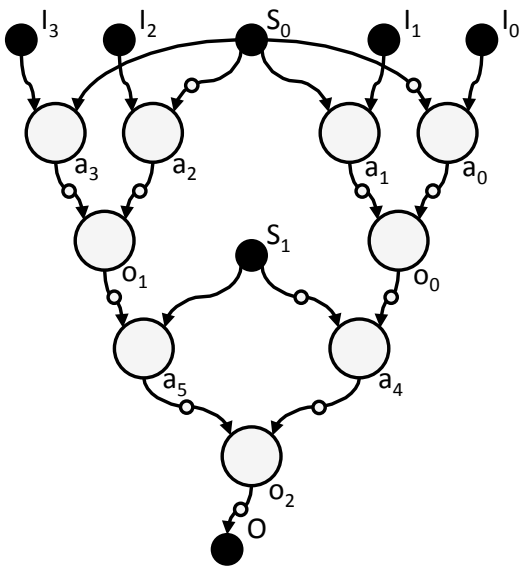


Figure 2.9: The And-Inverter Graph generated by the synthesis tool for the 4-to-1 multiplexer example. [18].

The most commonly used technology mapping algorithm uses a dynamic programming approach and searches for a depth-optimal covering of the logic circuit with cones. When mapping to K-LUTs, a cone is a subgraph of the logic circuit with K inputs and one output. Each cone can thus be implemented afterwards in a single LUT. The algorithm consists of three steps: cone enumeration, cone ranking, cone selection [24] [28]. The cone enumeration step starts at the input and recursively enumerates all possible cones until the outputs are reached. The second step also starts at the inputs, traverses the enumerated cones recursively and for each cone it calculates the depth, until the outputs are reached. The cone selection step starts at the outputs and selects the cones with minimal depth. It then recursively selects the cones at the input of the cone until the circuit inputs are reached. It can be shown that this algorithm is depth-optimal.

Several optimizations of the algorithm described above are possible. A common optimization is called area recovery, which tries to reduce the area, expressed as number of LUTs, without increasing depth [68]. Other optimizations try to reduce the run-time of the algorithm without compromising the quality of the resulting LUT circuit. The most common of such optimizations use pruning and priority cuts. When pruning is used, an upper-bound is set to the number of cones explored per node in the logic circuit [29]. The technique that uses priority cuts also avoids enumerating all cones, but attempts to enumerate a small number of cones (typically 5-10) that are likely to lead to good results [73]. Of course, when using these last optimizations, the algorithm is no longer guaranteed to be depth-optimal.

The algorithm described above can only handle combinational logic circuits. Sequential logic circuits, containing flip-flops (FFs), are first converted to combinatorial circuits. This is done by cutting the circuit through at the FFs, i.e. treating each input (output) of a FF as an output (input) of the combinational part.

Figure 2.10 shows the result of technology mapping for the 4-to-1 multiplexer example (in case of a 3-LUT technology target) and Figure 2.11 shows the resulting 3-LUT circuit.

2.2.3 Placement

The placement step assigns each abstract functional block, produced after technology mapping, to a physical block on the FPGA. A con-

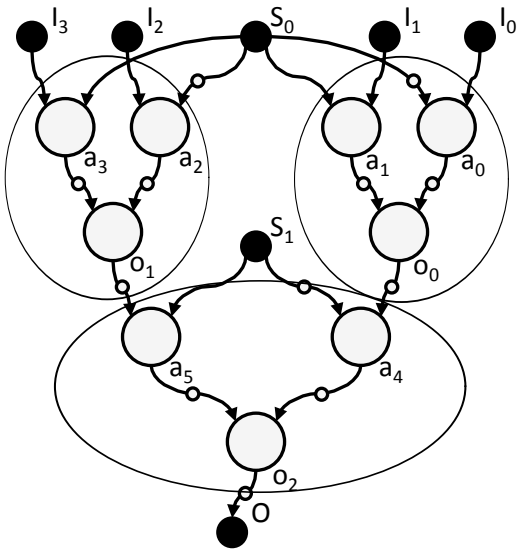


Figure 2.10: Result of the conventional technology mapping algorithm to 3-LUTs for the 4-to-1 multiplexer example.

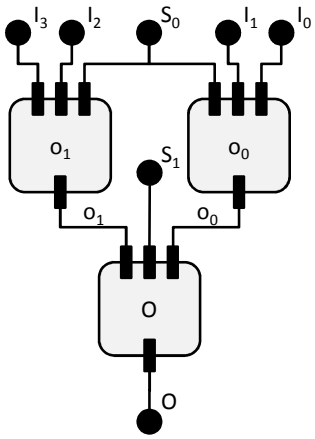


Figure 2.11: Resulting LUT circuit after conventional technology mapping to 3-LUTs for the 4-to-1 multiplexer example [18].

```
function place(Netlist NL, Architecture A):
  Placement P = randomPlacement(NL, A)

  while (T > T_min):
      move = findLegalMove(P)
      ΔC = costAfter(move, NL, P) − currentCost(NL, P)
      if (ΔC < 0 or uniform(0, 1) < e^(−ΔC/T)):
        P = acceptSwap(move, P)

  return P
```

Figure 2.12: Simplified pseudo code for a simulated annealing placer.

ventional FPGA placement algorithm takes two inputs: the circuit of functional blocks and a description of the FPGA architecture. The algorithm searches a legal placement for the functional blocks of the circuit that tries to reduce the number of wires that will be needed by the routing algorithm. In a legal placement every functional block is associated to (placed on) one of the physical blocks (without overlap).

The placement algorithm used in this dissertation, of which the simplified pseudo-code is shown in Figure 2.12, is based on simulated annealing [13] [57].

The algorithm starts by randomly, but legally, placing the functional blocks in the input circuit on physical blocks of the FPGA architecture. Then, the placer interchanges the functional blocks placed on two randomly chosen physical blocks in an attempt to improve the placement cost. If the move causes a decrease in placement cost, the move is always accepted. If on the other hand, the move causes an increase in placement cost the move is accepted with a probability of $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost due to the move. T is a parameter called the temperature, which controls the probability by which these moves, called hill-climbing moves, are accepted. Initially, T is very high so that most moves are accepted. Gradually T is decreased so that the probability by which hill-climbing moves are accepted decreases. When the temperature is decreased in the proper way the result is a low cost placement. The hill-climbing moves allow the placer to escape from local minima. Mostly the estimated

wire length needed by the router is used as a cost during placement.

2.2.4 Routing

A conventional router calculates the Boolean values that need to be stored in the memory cells of the configurable interconnection network so that the physical logic blocks are connected as specified by the nets in the mapped circuit. The main algorithm used to solve this problem is PATHFINDER [13] [71].

PATHFINDER presents the available routing resources of the FPGA in an easy-to-explore data structure, the routing resource graph (RRG). The RRG is a directed graph, where each node represents a routing wire on the FPGA and each directed edge represents a routing switch on the FPGA.[1]

An example of an RRG of a simple 2×2 island-style FPGA is presented in grey in Figure 2.13. The result after routing of the 4-to-1 multiplexer is shown in black.

In the PATHFINDER algorithm, the connections that need to be routed are organized in nets. These are sets of connections that share the same source. During the first routing iteration, nets can share resources at no extra cost and thus, each net is routed with a minimum number of wires. In subsequent routing iterations, the algorithm rips up and reroutes all the nets in the input circuit. A wire is said to be congested if it is used by more than one net. Wire congestion is not allowed in the final solution because this results in short-circuits. That is why the routing iterations are repeated until no shared resources exist or, in other words, the wire congestion is resolved. This is achieved by gradually increasing the cost of sharing resources between nets, a technique called *negotiated congestion*. The cost function of a wire in the RRG is

$$cost(n) = b(n) \cdot p(n) \cdot h(n), \tag{2.1}$$

where $b(n)$ is the base wire cost (equal to 1), $p(n)$ is the present wire congestion penalty and $h(n)$ is the historical wire congestion penalty.

The factor $p(n)$ is used to avoid wire congestion during one routing iteration. The factor $h(n)$ is used to make heavily used resources in past routing iterations more expensive. In this way a wire con-

[1]This is a simplification.The nodes can also represent logical pins or sources or sinks. These are treated in the same way [13].

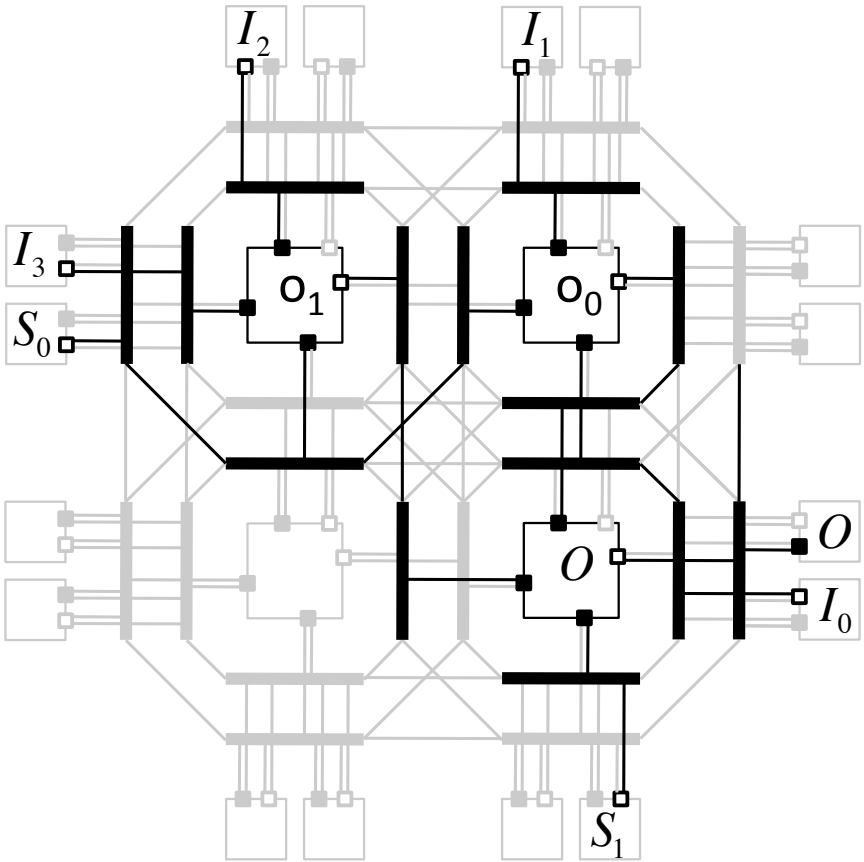


Figure 2.13: Implementation of the LUT circuit for the 4-to-1 multiplexer example in a simple 2×2 island-style FPGA resource graph (grey). Wires are solid lines; Edges are thin lines; Sources are open boxes; Sinks are filled boxes.

gestion map is built, which enables nets to avoid routing through heavily congested wires, if possible.

The present congestion penalty, $p(n)$, is updated whenever a net is rerouted. The update is done as follows

$$p(n) = \begin{cases} 1 & \text{if } c(n) > o(n) \\ 1 + p_f.(o(n) - c(n) + 1) & \text{otherwise} \end{cases} \tag{2.2}$$

where $c(n)$ represents the capacity of the node and $o(n)$ is the occupancy of the node. The capacity is the maximum number of nets that can legally use the routing resource. The occupancy of a node is the number of nets that are presently using it. The term $o(n) - c(n)$ thus represents the overuse of a node. The factor p_f is used to increase the sharing cost as the algorithm progresses. This is explained below.

The historical congestion penalty is updated after every routing iteration. The update is done as follows

$$h^i(n) = \begin{cases} 1 & \text{if } i = 1 \\ h^{(i-1)}(n) & \text{if } c(n) \geq o(n) \\ h^{(i-1)}(n) + h_f.(o(n) - c(n)) & \text{otherwise} \end{cases} \tag{2.3}$$

Again, the factor h_f is used to control the impact of the historical congestion penalty on the total resource cost.

Note that both the present and historical congestion mechanisms associate higher penalties with higher overuse.

The way the factors p_f and h_f change as the algorithm progresses is called the routing schedule. The routing schedule proposed in [13] is used. In this schedule, h_f is held equal to 1 independent of the iteration. On the other hand, p_f is initially set to 0.5 and is doubled in every subsequent iteration. More details on PATHFINDER can be found in [13].

2.3 FPGA configuration

FPGA tool flows generally don't generate a configuration, but a configuration bit stream. A configuration bit stream contains, besides the configuration data, the necessary instructions needed by the configuration controller to address the configuration infrastructure. There are three ways of configuring an FPGA: through an external configuration port, through an internal configuration port and using shift register LUTs (SRLs).

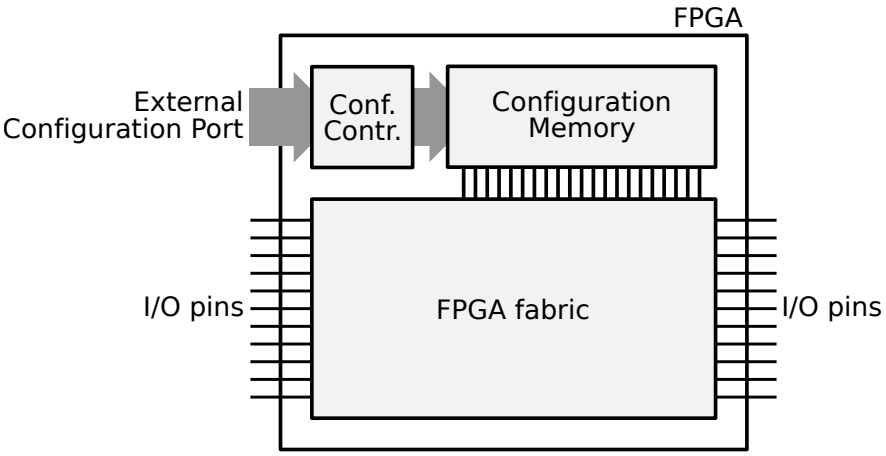


Figure 2.14: External Configuration of FPGAs [17].

2.3.1 External configuration

Configuration through an external port is the conventional and most used way of configuring an FPGA, shown in Figure 2.14. It is primarily used to configure the complete FPGA at start-up. The configuration bit stream is mostly read from a non-volatile memory chip such as an Electrically Erasable Programmable Read-Only Memory (EEPROM). After designing the hardware on a PC, the configuration bit stream generated by the FPGA tool flow is written to the EEPROM. This configuration method is less useful for Dynamic Partial Reconfiguration, as the configuration happens from outside the chip, making it necessary to have a desktop PC or an on-board embedded processor to perform the configuration.

2.3.2 Internal configuration

Modern FPGAs contain an Internal Configuration Access Port (ICAP), that makes it possible to configure the FPGA from within the FPGA. The ICAP uses the same protocol as the external configuration port. The HWICAP is a small piece of hardware that interfaces between the configuration manager and the ICAP. It is made using FPGA resources, it uses around a few hundred LUTs, a few hundred FFs and one BRAM.

The FPGA configuration memory is arranged as frames that are

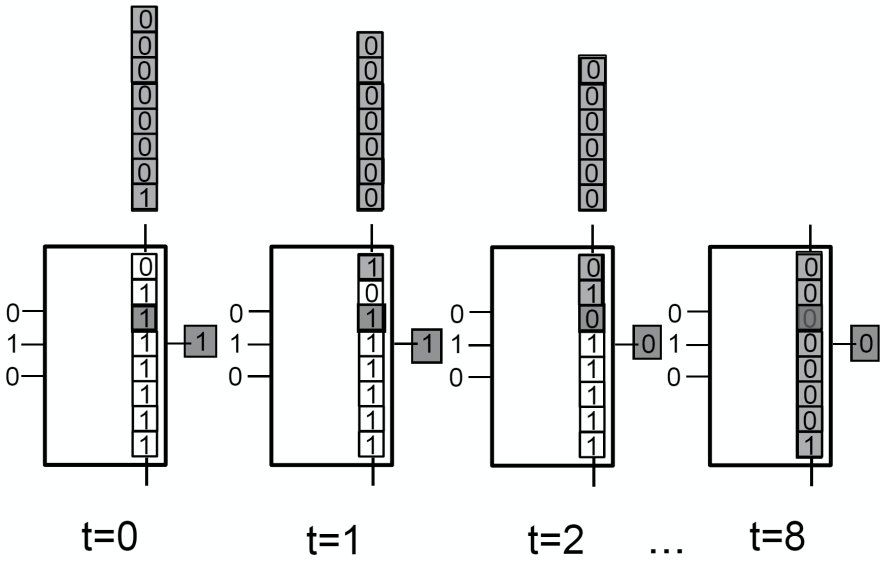


Figure 2.15: Reconfiguration of a 3-input shift register LUT.

tiled on the device. A frame is the smallest addressable segment of the configuration memory. For the Virtex 5 FPGA, for example, a configuration frame consists of around 1000 bits. All operations must therefore act upon complete configuration frames for both the external and internal configuration port. Frame-wise configuration can be inefficient since also configuration bits that do not change need to be rewritten as well.

2.3.3 SRL configuration

In Section 2.1.1 it was mentioned that the configuration bits of some LUTs of an FPGA, called shift register LUTs (SRLs), can be arranged as a shift register that is accessible from the configurable routing. It is important to note that the LUT still drives the output. An SRL is both shift register and LUT. The shift register functionality of an SRL can thus be used to (re)configure the LUT bits. This is illustrated in Figure 2.15, where a 3-LUT is reconfigured from a 3-input OR gate to a 3-input AND gate. The truth table bits of the LUT are numbered from 0 (binary 000) to 7 (binary 111). On the figure the input is 010, and therefore the third bit of the LUT drives the output. Note that, just as is the case with the previous configuration method, there is a

transient period where the FPGA contains a configuration that is not valid. In this example the reconfiguration takes 8 clock cycles, where the LUT is neither AND nor OR gate.

In order to make the truth table bits of multiple LUTs accessible from the configuration manager, they are grouped and each group is arranged as a larger shift register, called a reconfiguration path, by connecting the shift out of an SRL to the shift in of the next SRL. The shift in of the first SRL of each reconfiguration path is connected to a hardware block that interfaces to the configuration manager.

In Chapter 6 and Chapter 7 SRL reconfiguration is discussed in more detail and compared to ICAP reconfiguration.

Chapter 3

Dynamic Partial Reconfiguration

The configurability of FPGAs makes them a lot more flexible than ASICs, as the FPGA can be used for different implementations at different times. However, this conventional FPGA use is static: they are configured once at the start-up of the chip, after which the configuration is not changed during the operation of the application. Configuring an FPGA, however, is no more than writing the configuration memory. Since configuration memories are typically SRAM-based this happens relatively fast (in the order of tens of milliseconds for an average FPGA chip). When only a portion of the FPGA is reconfigured, it is referred to as Partial Reconfiguration. Some FPGAs can be reconfigured partially without interrupting the functionality of the rest of the FPGA. This is illustrated in the left-hand side of Figure 3.1, where a portion of the FPGA is reconfigured without interrupting a tranceiver that is also implemented on the FPGA. Because with partial reconfiguration only a portion of the configuration memory is rewritten, this allows for even faster reconfiguration. When Partial Reconfiguration is performed during the run-time of the application, the term Dynamic Partial Reconfiguration (DPR) is used. Different Xilinx FPGAs [50, 49] and the more recent 28 nm FPGA from Altera support DPR [32, 31].

Initially DPR was primarily used on a very large time scale, for bug fixes and system updates, since it allows for in-field updates without interrupting the system's operation. Recently DPR is also explored on a smaller time-scale. Indeed, with DPR it is possible to implement different functions, that are not needed at the same time

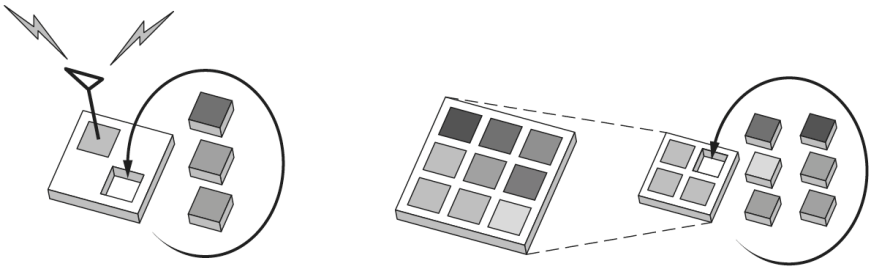


Figure 3.1: (left) With partial reconfiguration, part of the FPGA can be reconfigured without interrupting other logic present on the FPGA, in this case a transceiver. (right) DPR makes it possible to use smaller FPGAs, by time-multiplexing logic on the same FPGA region. [51]

in the system, on the same FPGA area. Due to hardware sharing, the amount of necessary resources will be smaller than in the case when each function uses a separate set of resources. The main reason to use DPR is therefore that it possibly implements an application using a smaller and thus cheaper FPGA. This is illustrated in the right-hand side of Figure 3.1. Besides increasing the area efficiency, DPR can possibly also improve speed and energy consumption.

The area on which the different functions are time-multiplexed is generally called the reconfigurable region (RR). Whenever one wants to change the implemented circuit, an amount of time is needed to rewrite the configuration memory of the reconfigurable region, called the reconfiguration time. This is an important aspect of DPR, because the interval between reconfigurations needs to be sufficiently long as to not nullify the gains obtained by performing DPR. In general, the overhead associated with the reconfiguration process is called the reconfiguration overhead. This can also entail the energy consumption and the hardware resources used by the reconfiguration process. The subsystem that performs the reconfiguration is called the configuration manager (CM) and is generally implemented in software. It is responsible for reprogramming the FPGA and might be responsible for generating a new configuration. This could require on-line computations or a simple lookup in an off-line generated database of configurations.

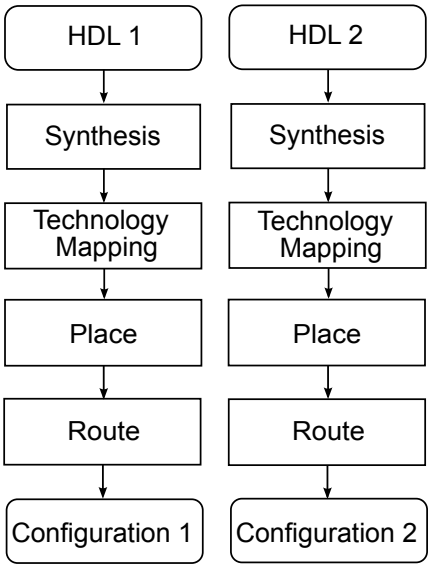


Figure 3.2: The conventional DPR tool flow.

3.1 Conventional DPR flow

The conventional DPR tool flow implements every circuit separately in the reconfigurable region by following the typical steps of an FPGA CAD flow (synthesis, technology mapping, placement and routing), as shown in Figure 3.2. The flow generates a configuration for each circuit. In the example in the figure, two configurations are generated, starting from a HDL description of each circuit. To switch between the different circuits the configuration manager overwrites the reconfigurable region with the appropriate configuration. The conventional DPR tool flow does not optimize for reconfiguration time and thus almost the complete reconfigurable region needs to be rewritten. This may lead to long reconfiguration times. For applications with a quasi-static behaviour it is previously shown that DPR can improve the efficiency. For more dynamic applications, however, this is no longer necessarily the case due to the relatively long reconfiguration time [75] [27] [67] [81] [74] [96].

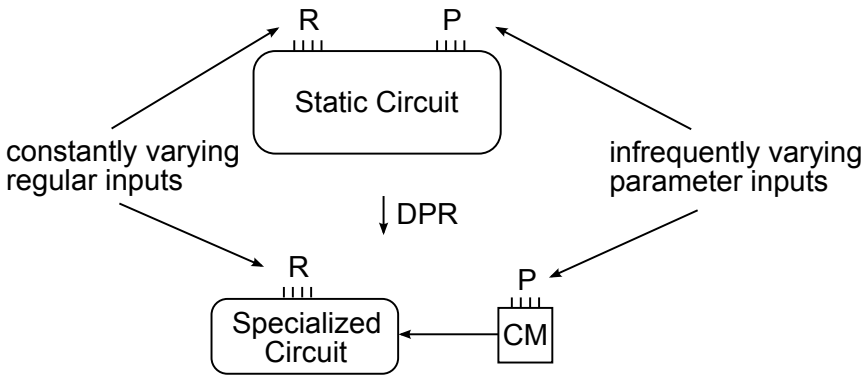


Figure 3.3: A DPR system for designs with infrequently varying parameter signals.

3.2 Dynamic Circuit Specialization

In certain circuits a subset of input signals, called parameters, change value only every now and then. An example of such a circuit is a constant multiplier in which one of the factors changes only once in a while and is thus considered constant between changes. In this case the constant factor is considered to be the parameter. These parameter inputs stay the same for a relatively long period and hence the state of the hardware driven by these signals also does not change. Only when the parameters change value, will the associated hardware driven by these signals change state. The hardware resources are therefore not used efficiently since part of the hardware is inactive most of the time.

DPR can offer a solution for this problem by moving the parameter inputs to the configuration manager (CM), as is shown in Figure 3.3. The CM is in this case responsible for generating a specialised circuit with only fast varying regular signals, whenever the parameter signals change value. The question then remains: how do we best set up such a DPR system?

A first possible solution uses constant propagation. When the parameters change value, the CM fills in the new values in a VHDL description, for example, and runs the complete FPGA tool flow to generate a new configuration. The circuit that is generated is possibly smaller and faster. The problem with this approach is that generating a configuration from a VHDL description takes a long time (at least in

the order of tens of seconds to minutes). This makes it less useful for more dynamic applications. Besides, this method is also not efficient since the complete tool flow is run for parts of the design that are not driven by the parameters.

Another approach could be to generate all the possible circuits off-line and store this in a database. The CM then only needs to look up the right configuration and reconfigure the FPGA. This is possibly the fastest approach. For some applications, however, an enormous amount of configurations are needed. Indeed the number of configurations grows exponentially with the number of parameter signals. Take the example of the constant multiplier. If the constant factor for example has a width of 32 bits, 2^{32} possible configurations need to be stored.

A third approach, called Dynamic Circuit Specialization, offers an elegant solution to this problem [20]. This method generates a configuration off-line that does not only contain constant 0's and 1's but also bits that are a Boolean function of the parameter signals. This type of configuration is called a parameterized configuration and the Boolean functions are called parameterized bits. Complex time consuming steps as placement and routing are all ran off-line. Whenever the parameters change value, the CM only needs to evaluate the parameterized bits and reconfigure the FPGA. Thus, generating a new configuration boils down to evaluating Boolean functions. This can happen orders of magnitude faster than running the complete FPGA tool flow. For the example of constant multipliers in [20], it is shown that this can be done while hardly reducing the quality of the implementation, in terms of area and speed, compared to complete constant propagation.

A tool flow that is able to automatically generate such parameterized configurations has been developed at the Hardware and Embedded Systems (HES) group at the University of Ghent. This tool flow is called the TLUT tool flow and is presented in more detail in the following section. It is available on-line at [43]. The TLUT tool flow generates parameterized configurations in which only the truth table bits are parameterized. A more general tool flow, called the DCS tool flow, that also generates parameterized routing bits is discussed in Section 3.2.2. The different steps of the DCS tool flow are still being researched. Although some prototypes of the different tools are already implemented, the complete DCS flow is not yet available.

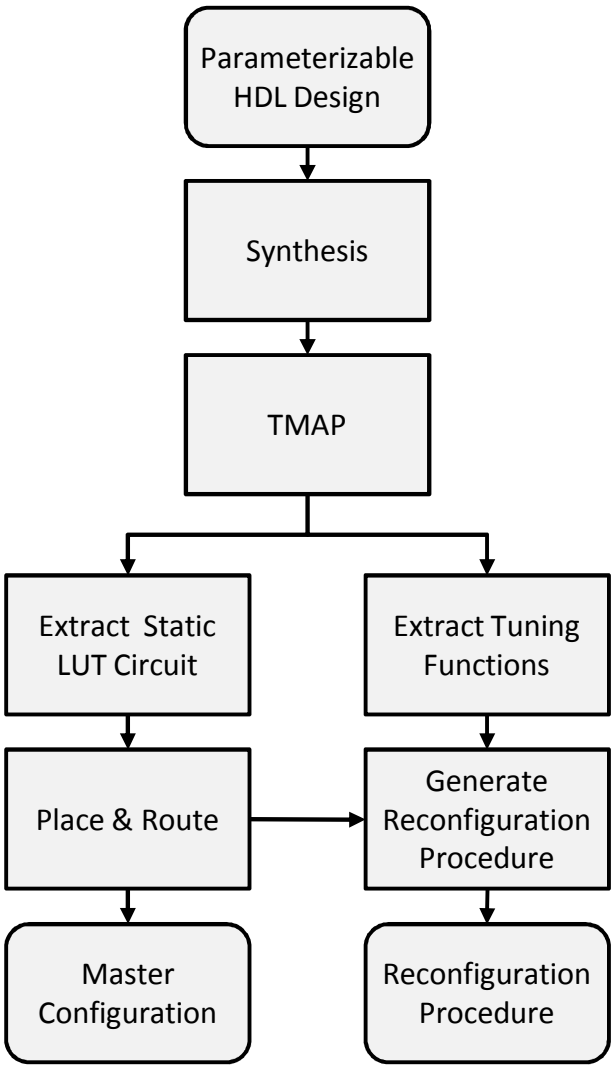


Figure 3.4: Overview of the TLUT tool flow [19].

```
entity multiplexer is
port (
  I   : in    std_logic_vector (3 downto 0);
  S   : in    std_logic_vector (1 downto 0); --PARAM
  O   : out   std_logic
);
end multiplexer;

architecture behavior of multiplexer is
begin
  O <= I(conv_integer(S));
end behavior;
```

Figure 3.5: The parameterized VHDL code of a 4-to-1 multiplexer where the select inputs are marked as parameters.

3.2.1 The TLUT tool flow

Figure 3.4 gives a detailed schematic overview of the the TLUT tool flow. Given an annotated VHDL description, it generates a master configuration and a reconfiguration procedure. The master configuration contains the constant portion of the configuration, with the regular, non-parameterized bits, together with the default values of the parameterized bits. It can be used to configure the FPGA at start-up. The reconfiguration procedure contains the necessary functions needed to specialize the FPGA's configuration when the parameters change value.

3.2.1.1 Annotated VHDL description

The TLUT tool flow starts from a VHDL description in which the parameter signals are annotated. In this section we will again use the example of a 4-to-1 multiplexer. Figure 3.5 shows the VHDL description of such a multiplexer. As can be seen on this figure, the selection signals are annotated as parameters. This is done by using the '–PARAM' annotation.

Just as a conventional synthesis tool, the synthesis step in the DCS tool flow converts the parameterized HDL description into a gate-level circuit. The logic circuit obtained for the multiplexer example is thus the same as the one shown in Figure 2.9. The only difference of

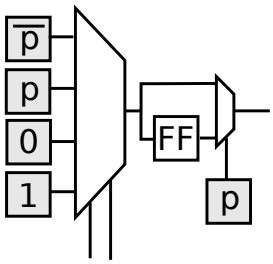


Figure 3.6: Example of a Tunable LUT, which is function of a parameter signal p.

the synthesis tool in the TLUT tool flow with the conventional flow is that it preserves the distinction between regular and parameter inputs.

3.2.1.2 TMAP

As discussed earlier, a conventional technology mapper generates a network of logic blocks, each consisting of a combination of a lookup table and a flip-flop. The truth table entries of the look-up table and the bit that controls the selection of the sequential output are constant zeros and ones. We will further refer to logic blocks simply as lookup tables or LUTs.

Figure 2.10 shows the result of the conventional technology mapping tool for the multiplexer example. The mapper looks for a depth-optimal covering of the logic circuit, with a minimum number of 3-LUTs. In this case three 3-LUTs are needed to implement the multiplexer. As can be seen on the figure the mapper looks for parts of the logic circuits with maximum three inputs and one output, as such a subcircuit can be implemented with a 3-LUT. Figure 2.11 and Figure 2.13 show the resulting LUT circuit and its implementation after placement and routing, respectively.

The heart of the TLUT tool flow is an adapted technology mapper. This mapper does not map to regular LUTs, but to Tunable LUTs or TLUTs. A TLUT is a LUT of which the truth table bits are expressed as a Boolean function of the parameter signals. In Figure 3.6 an example is shown of a 2-input Tunable LUT, which is function of a parameter signal p. The adapted technology mapper is called TLUT mapper or TMAP.

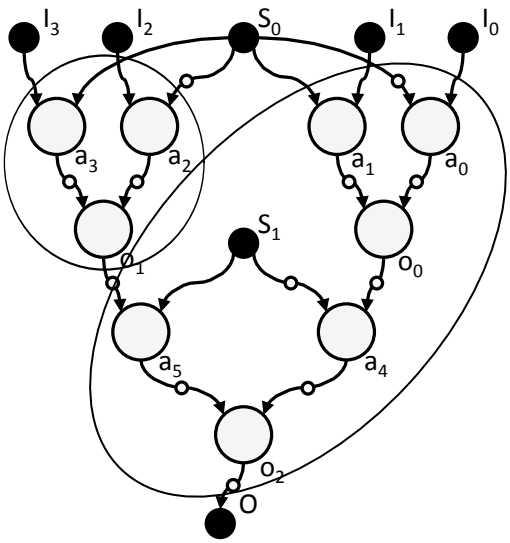


Figure 3.7: Result of the technology mapping using TMAP for the 4-to-1 multiplexer example.

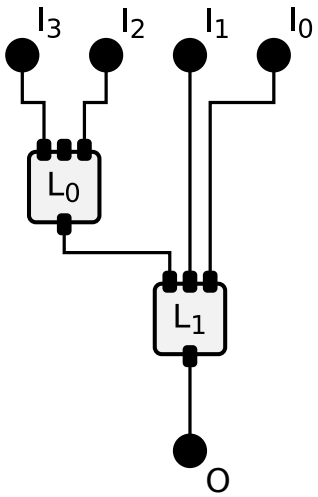


Figure 3.8: Resulting LUT circuit after technology mapping for the 4-to-1 multiplexer example, when using TMAP [18].

The result of TMAP for the 4-to-1 multiplexer example is shown in Figure 3.7. Because the synthesis tool in the TLUT tool flow preserves the distinction between regular inputs and parameter inputs, TMAP is aware of which signals are infrequently varying. This is taken into consideration when searching for a covering with LUTs. TMAP searches for subcircuits with maximum three regular inputs, one output and any number of parameter inputs. Such a subcircuit can be implemented with a 3-input TLUT. The presence of parameter signals thus translates into a LUT that is dynamically reconfigured when the parameter signals change value.

The resulting LUT circuit obtained by TMAP is shown in Figure 3.8 and only uses two 3-LUTs. It is thus more area efficient compared to the conventional mapper, which needs 3 LUTs to implement the 4-to-1 multiplexer.

3.2.1.3 Generating the master configuration

Only the truth table contents of the LUT circuit obtained by TMAP are parameterized, the connections between the LUTs are fixed. This static LUT circuit can thus be placed and routed using the conventional tools. An example of the implementation of the TLUT circuit on a two by two FPGA is shown in Figure 3.9. Note that besides using less LUTs, also less routing is used compared to the conventional implementation in Figure 2.13.

Because the code of the technology mapper in the Xilinx tool flow is not open-source, a workaround is used to implement TLUT circuits on Xilinx FPGAs. This workaround uses VHDL library primitives that instantiate LUTs. The TLUT circuit is thus written out in VHDL and processed by the complete Xilinx flow.

After routing, the master configuration can be easily extracted, just as in the conventional FPGA flow. This configuration can be used to configure the FPGA at start-up.

Xilinx provides intermediary formats in the tool flow that can be used to change the implementation of a design. After technology mapping, placement and routing a Native Circuit Description (NCD) file is generated. This NCD file can be converted to the human-readable Xilinx Design Language (XDL) format. To add the appropriate truth table values for the desired default parameter values adjustments are made in the XDL file after routing. The adjusted XDL file is then converted back to an NCD file and the associated bit stream.

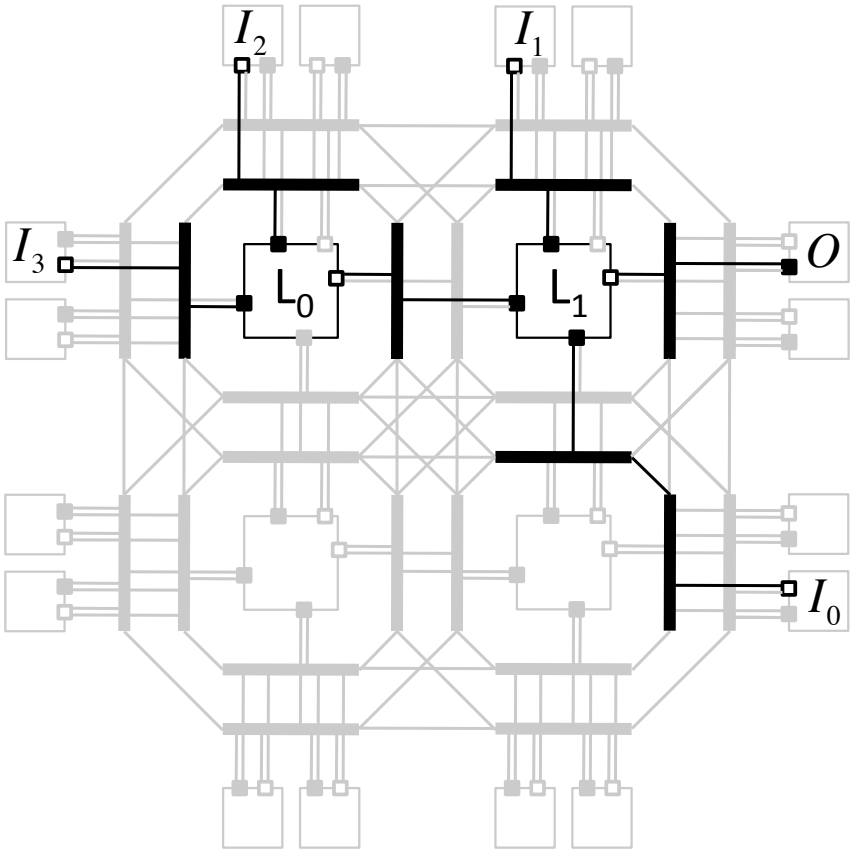


Figure 3.9: Implementation of the TLUT circuit for the 4-to-1 multiplexer example in a simple 2×2 island style FPGA resource graph (grey). Wires are solid lines; Edges are thin lines; Sources are open boxes; And sinks are filled boxes.

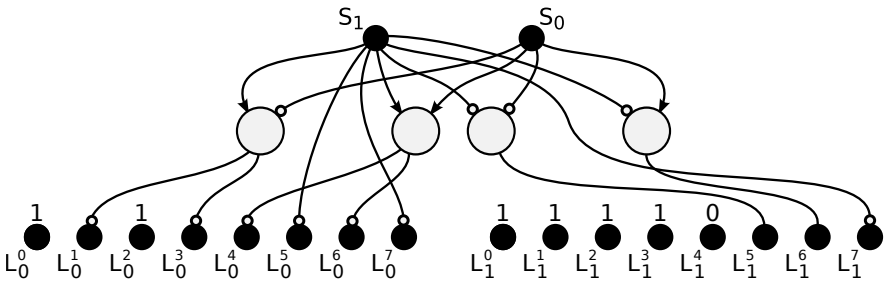


Figure 3.10: The Partial parameterized Configuration of the 4-to-1 multiplexer example [18].

3.2.1.4 Generating the reconfiguration procedure

Besides a LUT circuit, TMAP also generates a Partial parameterized Configuration (PPC). This is a Boolean circuit that expresses the truth table contents of the TLUTs as a function of the parameters. The PPC of the multiplexer example is shown in Figure 3.10. This PPC is automatically compiled into C code that can be used to generate the specialized truth table contents of the TLUTs each time the parameter signals change value. The evaluation C function of the PPC of the 4-MUX is shown in Figure 3.11.

The reconfiguration procedure also contains the functions needed to address the configuration interface. In the original implementation of the TLUT tool flow the HWICAP is used to interface to the ICAP and reconfigure the FPGA. The `XHwIcap_SetClbBits` function provided by Xilinx is used. This function takes the location of the LUT to be reconfigured as input. To be able to reconfigure the TLUTs using this function, their location is extracted from the XDL file after routing.

3.2.1.5 Benefits of the TLUT tool flow

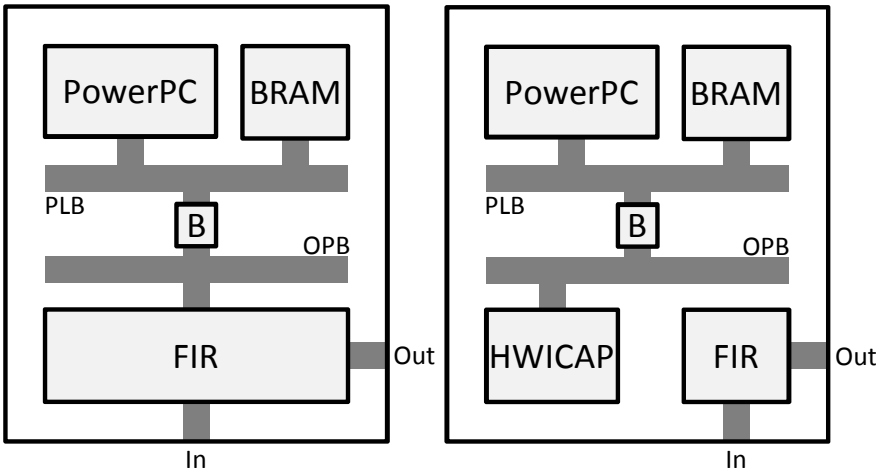
In [18] the TLUT tool flow was implemented on a Xilinx platform to illustrate the benefits compared to a static implementation. The XUP board was used that contains a Virtex 2 Pro (XC2VP30) FPGA, with 30000 LUTs. The Virtex 2 Pro FPGA also contains two PowerPCs. The Configuration Manager (CM) is implemented in software on one of the PowerPCs. The CM communicates with the HWICAP to reconfigure the FPGA when the parameter signals change value.

```
void evaluatePPCMux (int s[], int L[][])
  int  node0 =  s[0]  & !s[1];
  int  node1 = !s[0]  & !s[1];
  int  node2 =  s[0]  &  s[1];
  int  node3 = !s[0]  &  s[1];

  L[0][0]  = 1;
  L[0][1]  = ! node3;
  L[0][2]  = 1;
  L[0][3]  = ! node3;
  L[0][4]  = ! node2;
  L[0][5]  = ! s[1];
  L[0][6]  = ! node2;
  L[0][7]  = ! s[1];

  L[1][0]  = 1;
  L[1][1]  = 1;
  L[1][2]  = 1;
  L[1][3]  = 1;
  L[1][4]  = 0;
   L[1][5]  = node1;
  L[1][6]  = node0;
  L[1][7]  = ! s[1];
```

Figure 3.11: The C-function to evaluate the PPC of the 4-to-1 MUX.



(a) Static implementation (b) DCS implementation

Figure 3.12: Static implementation of the FIR filter (a), compared to the DCS implementation (b) [18].

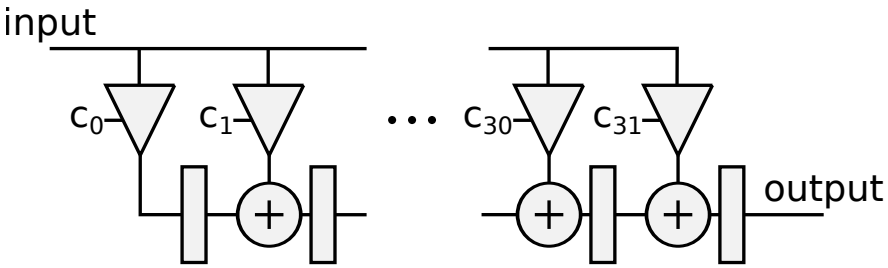


Figure 3.13: Schematic representation of the 32-tap FIR filter [18].

The platform implementations used in case of static and DCS implementation are shown in Figure 3.12 (a) and Figure 3.12 (b), respectively.

The tool flow is used to implement a 32-tap Finite-Impulse Response (FIR) filter with an 8-bit input and 8-bit coefficients, shown in Figure 3.13. The static implementation of the FIR filter is shown in Figure 3.12 (a). The coefficients of the filter are changed by the PowerPC by writing to registers. In Figure 3.12 (b) the implementation using the TLUT tool fow is shown. In this case the coefficients of the FIR filter are chosen as parameters. Every time the coefficients change value, the CM will reconfigure the FPGA using the reconfig-

Table 3.1: Comparison of the static implementation and the DCS implementation of the FIR filter [19].

	Static	DCS
FIR Area (LUTs)	4,259	1,985
System Area (LUTs)	1,218	1,298
Total Area (LUTs)	5,477	3,283
Reconf. time (ms)	N/A	151

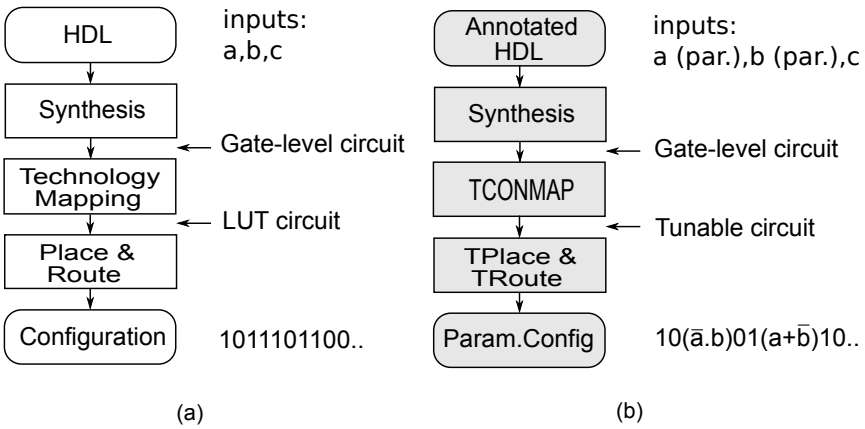


Figure 3.14: (a) Conventional FPGA tool flow (b) DCS tool flow.

uration procedure. Both implementations are tested by comparing the values generated in hardware to values simulated in software.

The results obtained for both implementations are shown in Table 3.1. The results show that using the TLUT tool flow the adaptive filtering application can be implemented using 40% less resources[19]. However, the static implementation can change the coefficients in a few clock cycles. In this implementation, the parameterized implementation needs 151 ms to evaluate the parameterized bits and reconfigure the FPGA.

3.2.2 The DCS tool flow

The general DCS tool flow is shown in Figure 3.14 (b) and is compared to the conventional FPGA tool flow in Figure 3.14 (a). This DCS tool flow also takes in a HDL description in which the designer annotates the infrequently varying inputs as parameters. The main difference with the TLUT tool flow is that it generates a configuration in which both the LUTs and routing are parameterized.

The core of the TLUT tool flow, the technology mapper TMAP, maps to a circuit with Tunable LUTs or TLUTs. The TCONMAP technology mapper used in the DCS flow, on the other hand, maps the parameterized design onto a Tunable circuit [46]. This is a network of TLUTs interconnected with Tunable Connections (TCONs). These connect a source and a sink, as do regular connections, but each Tunable connection is also associated with a Boolean expression that is called the activation function. A Tunable connection only needs to be realised for the parameter values for which the activation function evaluates to true. Note that a Tunable circuit can contain connections that have the same sink as long as their activation functions never evaluate to true at the same time. Figure 3.15 presents an example of a Tunable circuit, implemented with 2-LUTs. When the parameter signals in a Tunable circuit are associated with a concrete binary value, it results in a regular LUT circuit. For the example in Figure 3.15 the associated LUT circuits are shown for the parameter values $p = 0$ and $p = 1$.

TPlace and TRoute, the adapted placer and router in the DCS tool flow, can further refine the Tunable circuit to a parameterized configuration in which some of the configuration bits are expressed as Boolean functions of the parameters [21, 93, 94, 95], namely the parameterized bits.

3.3 Limited-context Dynamic Partial Reconfiguration

In theory, the number of circuits that can be time-multiplexed in a reconfigurable region is only limited by the size of the memory available to store the different configurations associated with the circuits. In many cases, however, only a limited number of circuits are needed. In this work we will call this special case of DPR, limited-context DPR. Figure 3.16 shows an FPGA with a reconfigurable re-

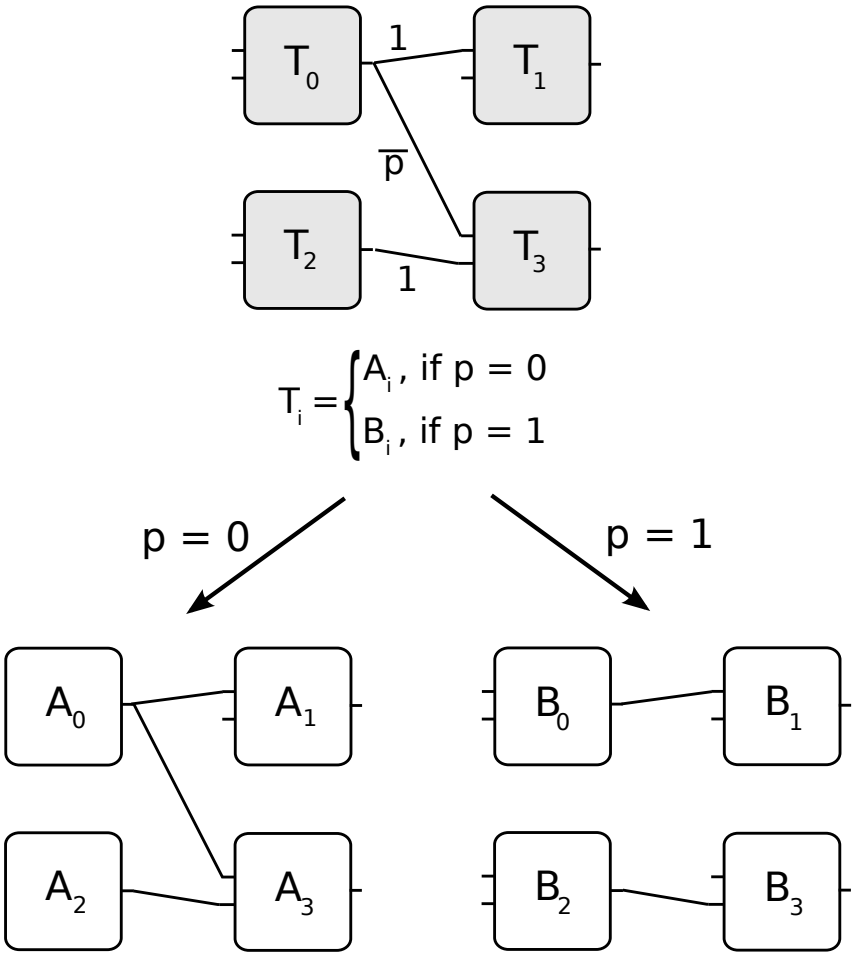


Figure 3.15: A Tunable circuit and its associated logic circuits.

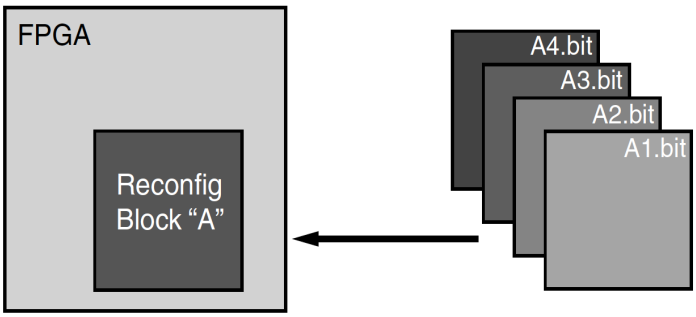


Figure 3.16: Example of limited-context DPR with four contexts or circuits [52].

gion that contains reconfigurable block A. There are only 4 versions of block A between which is switched during run-time using DPR. An example that uses limited-context DPR is a mobile transceiver that supports different communication standards (like CDMA or Wi-Fi), but only uses one at any given time. In this case, every context is a circuit that contains the necessary functions to support the corresponding communication standard. Another example is an audio application which only contains a low-pass and a high-pass adaptive filter.

As mentioned in the introduction, the conventional DPR flow generates a configuration for each circuit by implementing each circuit separately. The conventional DPR flow does not optimize the reconfiguration overhead. In this dissertation I will show that the reconfiguration overhead can be reduced through a combined implementation of the limited number of circuits at a certain point of the tool flow.

3.4 Related work

In [85] the modular design flow of Xilinx is described. This is the basic flow that is used to implement dynamic partial reconfiguration

on Xilinx FPGAs. It is the same as the conventional DPR flow described in this dissertation. It runs the tool flow separately for each circuit and therefore reconfigures almost the complete reconfigurable region. It does not try to minimise reconfiguration overhead such as in my work. Xilinx also provides a difference-based partial reconfiguration flow [40]. In this flow a designer is able to manually make low-level changes to one design, after which the flow generates a partial bitstream that incorporates only these changes. In contrast to the difference-based flow, my work on limited-context DPR considers several circuits being time-multiplexed on the same region in an automatic way.

An approach to generating specialized configurations for DPR is running the conventional FPGA tool flow on-line. This is only useful for quasi-static implementations, since running the complete tool flow takes considerable time. Some authors propose to improve this approach by using simpler versions of the FPGA tools. This way the quality of the implementation is traded off to reduce the overhead of the CM. A first example of this approach is the work done in [62, 87]. In this work generic netlists are created which are already partially evaluated for certain parameter signals. This way the synthesis step is avoided. The time consuming placement and routing steps, however, still need to be performed on-line.

Some authors focus on specific designs that are manually optimized. In [100] [99], for example, a constant multiplier in which the constant can be specialized during run-time is hand-designed and optimized. Although manually designed implementations can be efficient, the required design effort is very high.

Considerable research has been done on reducing the reconfiguration time of dynamic partial reconfiguration. Several authors consider changing the configuration infrastructure of the FPGA to speed up the configuration process. The authors in [42], [91], [92] propose a time-multiplexed FPGA in which each configuration memory cell is backed by 8 bits of inactive storage in the configuration SRAM. This architecture was specifically designed for rapid switching between a limited number of contexts. In [69] the authors extend a regular configuration memory with a barrel shifter. This architectural enhancement does not only speed up reconfiguration, but also the relocation of portions of the configuration memory. In [35] an extension is proposed to the configuration infrastructure that allows to directly handle compressed bitstreams. In contrast to these works,

our novel tool flow does not require any changes to the configuration infrastructure. The FPGA only needs to support dynamic partial reconfiguration.

Several publications, such as [30] [37] [59] [86], focus on reducing the reconfiguration time on a higher abstraction level in the tools, namely the one of modules or task graphs. They address the problem of scheduling several circuits on several reconfigurable regions in such a way that execution time is optimized. As main objective they try to overlap the configuration time and calculation time of tasks. The possibility of changing the location of a module or task, a concept called relocation, is also investigated in this context [12] [54]. In [48] relocation is considered in a heterogeneous architecture.

Another approach is expediting the actual reconfiguration process. In [64] and [39] this is done by using configuration pre-fetching. In this case, the configuration bit stream is partly loaded to on-chip memory before the actual reconfiguration process starts. This is of course only possible in applications where the new configuration is known beforehand.

Several authors show that the hardware that interfaces the internal reconfiguration port, the Hardware Internal Configuration Access Port (HWICAP), can be sped up significantly [26] [39] [41]. The techniques considered are Direct-Memory-Access and overclocking. In [26] custom circuitry is added to detect whether reconfiguration was successful. In [47], it is shown that the inclusion of the configuration access port into the data path of a processor core using a Fast Simplex Link (FSL), instead of the Processor Local Bus (PLB) interface to the HWICAP, results in a speed up of the reconfiguration process.

When the number of circuits that are time-multiplexed on the same FPGA region is limited, combined optimization approaches can be considered to reduce area. In [23], the authors use a high-level tool, called GAUT [33], for a combined implementation of different data flow graphs of digital signal processing applications, by maximizing the similarities between control steps. They do not use dynamic partial reconfiguration, but add multiplexers where necessary. In [82] the authors attempt to increase the correlation between the configurations by placing the LUTs of the different circuits in such a way that the connections between the LUTs overlap. A technique they call edge matching. The overlapping connections can be implemented using the same routing resources and thus the correla-

tion between the different routing configurations is increased. They do, however, not consider the organization of configuration bits in frames. Also, further on in this work it is shown that edge matching increases the total wire length of the circuits considerably [4].

Others focus on increasing the correlation between the truth tables of LUTs that occupy the same physical LUT [25] [77] [78]. This is done by changing pins to which the LUT inputs connect and taking advantage of don't-care values. In [25] the placement is adapted to further increase the correlation of the LUTs. Increasing the correlation between LUTs can only have a limited impact when trying to reduce the reconfigurations time, as most configuration bits control how the FPGA's interconnection network is configured. In this work, for example, the contribution of CLBs (that contain several LUTs) to the total configuration is found to be around 30%. Also, a static CLB frame is only generated when all the bits of that frame correspond for the different circuits.

There is also previous work that focuses on reducing the number of frames that need configuration. In [65] it is proposed to adjust the router for this purpose. This is done to reduce the load time of applications on FPGAs in general, it does not consider dynamic partial reconfiguration. The work in [90] adjusts the placer to reduce the number of frames and also considers dynamic partial reconfiguration. However, they do not consider increasing the correlation between the routing configurations of the different circuits as in my research. The approach taken is also reducing the size of the bit stream of each of the circuits separately.

More recent research in literature on DPR focuses on applying the technique to different applications such as: face recognition [2], sql query execution [36] [11], triple modular redundancy [22] and regular expression matching [97]. Also, some work considers using DPR on the Zynq platform [6] [79] [58].

The work done in my research is complementary to the work done about extending the regular configuration infrastructure, high-level task scheduling and expediting the actual reconfiguration process. It suits very well the work done about scheduling where a portion of the FPGA is used to time-multiplex different tasks. The tools I developed could be used to significantly decrease the time needed to switch between tasks in one reconfigurable region.

Chapter 4

Exploring novel tool flows for limited-context dynamic partial reconfiguration

The conventional DPR flow implements the different contexts or circuits separately in the reconfigurable fabric, independent of the number of circuits. This is illustrated in Figure 3.2 for 2 circuits. The conventional DPR flow does not consider the reconfiguration overhead, making DPR less useful for more dynamic applications. In contrast, in this chapter the possibilities and limitations are explored of novel tool flows that use a combined approach to implement a limited number of circuits, thereby reducing the reconfiguration overhead.

The configuration memory consists mostly of bits that control the programmable interconnection network. That is why the focus in this chapter is on reducing the reconfiguration overhead of the interconnection network. We therefore assume all Look-up Table (LUT) bits are rewritten and the tool flows explored in this chapter try to reduce the number of routing bits that need rewriting.

The main idea in this chapter is, given a limited number of circuits represented by a high level HDL representation, to form a single tunable circuit. This tunable circuit contains all circuits as different implementations of the tunable circuit that are mutually exclusive in time. Using tools from the DCS flow the tunable circuit can then be further implemented in the reconfigurable fabric.

TRoute, the router used in the DCS flow, plays a central role in

the tool flows explored in this chapter. In the first section, TRoute is therefore presented in more detail. The second section discusses three tool flows I developed that each perform a combined implementation of the circuits, taking the reconfiguration overhead into consideration. The first tool flow uses edge matching, which was explained earlier in Section 3.4. When I started my research this was the only technique I found in literature that also considered a combined implementation approach to reduce the reconfiguration overhead. The second tool flow optimizes the total wire length of the tunable circuit during placement. In the last tool flow the wire lengths of each of the different different circuits are optimized separately during placement. These three tool flows are first presented separately and then compared. This comparison was done based on experiments with tunable circuits containing only two circuits. The final section of this chapter researches the limitations and benefits of a combined implementation approach when the number of circuits increases.

4.1 TRoute: a connection-based DCS router

As we explained in Section 2.2.4, a conventional router calculates the Boolean values that need to be stored in the configuration bits of the configurable interconnection network so that the physical logic blocks are connected as is specified by the nets in the mapped circuit. The main algorithm used to solve this problem is PATHFINDER [71]. The PATHFINDER algorithm was discussed in more detail in Section 2.2.4. PATHFINDER presents the available routing resources of the FPGA in an easy-to-explore data structure, the routing resource graph (RRG). In the PATHFINDER algorithm the connections that need to be routed are organized in nets. These are sets of connections that share the same source. Connections of a same net are allowed to share resources. In every routing iteration, the algorithm rips up and reroutes all the nets in the input circuit. These iterations are repeated until no shared resources exist between nets or, in other words, the routing trees of the nets are disjoint. This is achieved by gradually increasing the cost of sharing resources between nets, a technique called negotiated congestion. The cost function of a routing node n in the RRG is

$$cost(n) = b(n) \cdot p(n) \cdot h(n), \tag{4.1}$$

where $b(n)$ is the base cost, $p(n)$ is the present congestion penalty and $h(n)$ is the historical congestion penalty.

TRoute is the reconfiguration-aware router used in the Dynamic Circuit Specialization (DCS) tool flow presented in [93]. This DCS tool flow takes in a design with infrequently varying inputs, called parameters, and generates a parameterized configuration. This is an FPGA configuration that expresses the configuration bits as a Boolean function of the parameters. Whenever the parameters change value, the Boolean functions are re-evaluated and written in the configuration memory of the FPGA.

An important concept in DCS is a tunable connection (TCON). This is a connection, with which a Boolean function of the parameters is associated, called the *activation function*. The connection between the source and the sink of the TCON only needs to be realized in the FPGA fabric for parameter values for which the activation function evaluates to $True$. In contrast to nets, TCONs can legally share a node in the RRG. This is allowed when they have the same source or when they have the same sink. TCONs that share the same sink are not active at the same time, have disjoint activation functions and can therefore share routing resources.

TRoute is based on the PATHFINDER algorithm and is developed to route a set of tunable connections [21][93][94]. Instead of nets, TRoute rips up and reroutes tunable connections. The cost function of a node n in the RRG, in the case of TRoute, is

$$cost(n) = \frac{b(n) \cdot p(n) \cdot h(n)}{share(n)}, \tag{4.2}$$

where $b(n)$, $p(n)$, $h(n)$ are as in Equation 4.1 and $share(n)$ is the number of TCONs that legally share a node.

Note that this equation does not necessarily penalize historical sharing between connections of the same net. To calculate the occupancy, it is not the case that the different connections that use a node are counted. Instead a partition is made of the connections. The connection are divided into groups such that connections in one group can legally share routing resources. Connections can legally share resources if they have the same source or if they have the same sink. Only two partitions are considered, one based on the sources and one based on the sinks. The partition with the minimum nuber of groups is chosen. This minimum number of groups is considered to be the occupancy. So the connections that legally share a routing

node are not counted in the occupancy. Also the legal sharing in the other groups of the partitions is not counted and is therefore not penalized. More details can be found in [21][93][94].

In Figure 4.1(a) an example is shown of a set of tunable connections that implement the functionality of a four way switch. The straight connections are realized when $p = 0$, the crossed connections when $p = 1$. In Figure 4.1(b) the implementation is shown of this set of TCONs, as generated by TRoute on a 2×2 island-style FPGA. To generate the parameterized configuration bits of the switches the activation functions of the TCONs that use the switch are logically added (OR). Note that a lot of static ones are generated. This is because TRoute stimulates sharing of resources between connections with disjoint activation functions. In this example only eight configuration bits are parameterized or dynamic, the rest is static. It is straight-forward to generate a specialized configuration corresponding to one of the connection patterns of the four-way switch. Only the parameterized bits need to be evaluated.

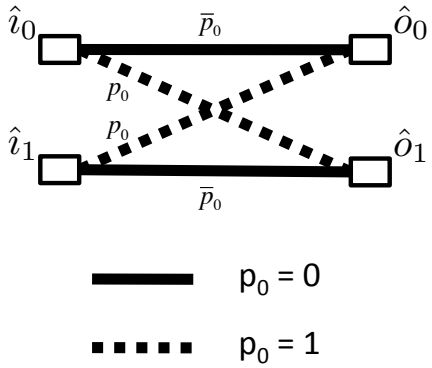
4.2 Exploration of novel tool flows

In this chapter we explore tool flows that combine several LUT circuits into a tunable circuit. This is illustrated in Figure 4.2, in which two LUT circuits A and B are combined into a tunable circuit T. It can be verified that T implements A when $c_0 = 0$ and B when $c_0 = 1$. The tunable circuit can be further implemented in the reconfigurable fabric using the tools from the DCS flow, such as TRoute. Merging of several LUT circuits into a tunable circuit consists of two steps:

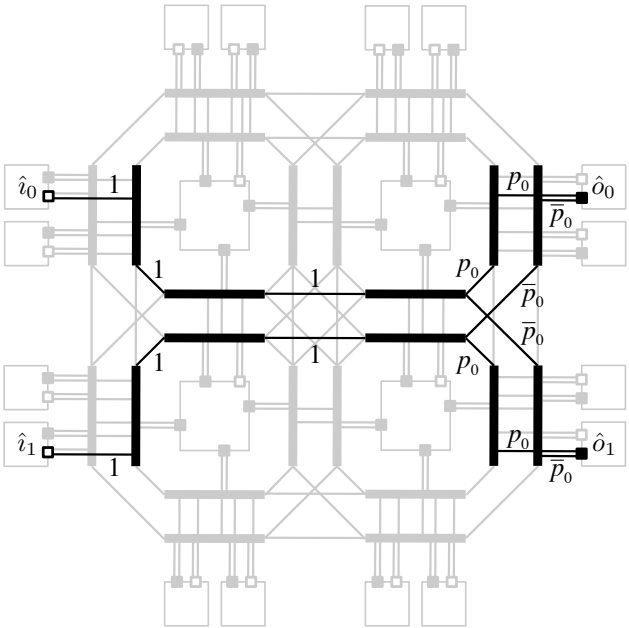
1. determine which LUTs will be implemented using the same tunable LUT;

2. the annotation of the connections with the appropriate activation function to generate the tunable connections.

Indeed, we essentially have one degree of freedom when generating a tunable circuit. We have to determine which LUTs will be implemented using the same tunable LUT. Of course, only LUTs belonging to different circuits can be combined in the same tunable LUT. Once this is decided, generating the parameterized bits of the tunable LUT is very straight-forward, as is shown in the example in Figure 4.3.



(a)



(b)

Figure 4.1: (a) Schematic representation of a set of TCONs with the functionality of a four-way switch. (b) Implementation of that set of TCONs (black) in a simple 2×2 island style FPGA resource graph (grey). Wires are solid lines; Edges are thin lines; Sources are open boxes; And sinks are filled boxes.

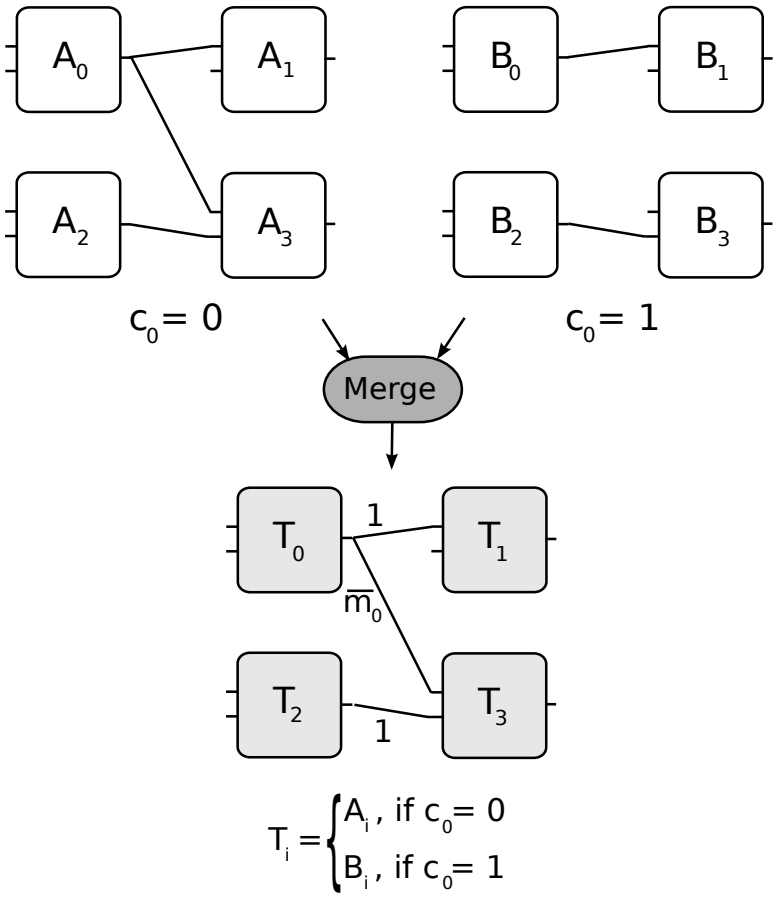


Figure 4.2: Merging two LUT circuits into a tunable circuit.

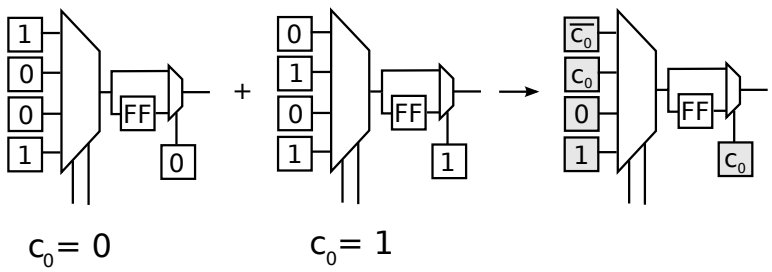


Figure 4.3: Generating parameterized tunable lookup table bits.

We first assume the circuits are numbered and express this number in a binary fashion. For example, if there are three circuits, we need two bits c_1c_0 to express the circuit number. Every circuit then corresponds to a Boolean product that evaluates to True for the appropriate circuit. For example, when the circuit c_1c_0 is *10* the Boolean product is $c_1.\overline{c_0}$. The bits of a LUT are first multiplied (AND) with the Boolean product of the circuit the LUT belongs to. The corresponding bits of the different LUTs are then added (OR) to generate the Boolean expressions that represent the parameterized bits of the tunable LUT. For example, for the highest bit of the truth table in Figure 4.3 we get $\overline{c_0}.1 + c_0.0$ which simplifies to $\overline{c_0}$. We note that when evaluating the tunable LUT on the right for a certain circuit, the correct bit values for the LUTs on the left are obtained. Using the method above, a tunable LUT can implement any combination of LUTs, as long as they belong to different circuits.

The topology of the tunable circuit is determined, once it is decided which LUTs are implemented in the same tunable LUT. The connections initially connecting the LUTs will simply connect the corresponding tunable LUTs. To generate the tunable connections, the connections of all the circuits are annotated with an activation function that consists of the Boolean product that corresponds to the circuit the connection belongs to. When connections have the same source and sink they can be merged into one tunable connection of which the activation function is an addition of the Boolean products of the connections. An example is given in Figure 4.2. In this figure, connections that are used in both circuits have as activation function $\overline{c_0} + c_0$ which simplifies to *True* or *1*.

In Figure 4.2 we simply implement the LUTs with the same index using the same tunable LUT. There are however many ways to combine the different LUTs in one tunable LUT, each generating a tunable circuit with a different topology. In what follows, we present three different tool flows that generate a tunable circuit, each using its own optimization criterion.

4.2.1 Edge matching

The first tool flow we propose, as shown in Figure 4.4, takes in the HDL descriptions of the different circuits and generates a parameterized configuration, in which some of the bits are expressed as a boolean expression of the circuits, for example $1, 0, 0, \overline{c_1}.c_0, \overline{c_0}, 1, 0$....

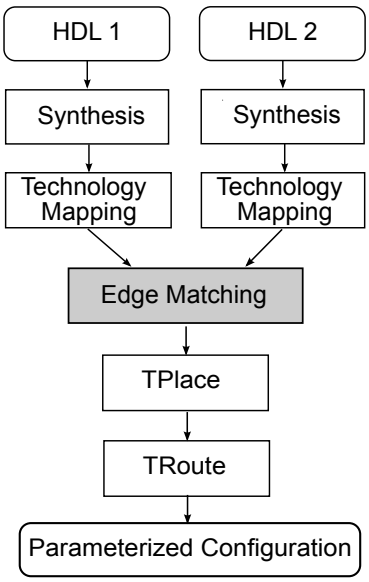


Figure 4.4: Tool flow that uses edge matching when merging LUT circuits into a tunable circuit.

The tool flow is clearly a combination of the conventional and DCS tool flows. The conventional tool flow is followed up until the technology mapping, thus generating a circuit of LUTs for every input HDL description. In the following step the LUT circuits are combined into one tunable circuit that is further implemented in the reconfigurable region using the TPlace and TRoute step of the DCS tool flow. The proposed tool flow thus reuses much of the steps of the conventional and DCS tool flows. The key step in the tool flow is the combination of different LUT circuits into one tunable circuit, as is shown in Figure 4.2.

In this first tool flow a technique was used that is found in literature as *edge matching*. Edge matching tries to reduce the number of tunable connections in the tunable circuit formed. It does this by combining LUTs into tunable LUTs in such a way that the number of connections that have the same source and sink is maximized. These overlapping connections can be implemented by the same physical connection. It is obvious that when the application changes between these circuits no switches need to be turned in the routing for this connection. Edge matching attempts to reduce the reconfiguration

overhead by reducing the number of switches that need reconfiguration.

Circuit edge matching was first proposed in [82]. However, these authors did not have a router that could route the wires of the different circuits simultaneously. In addition, no results regarding the impact on the quality of implementation of the circuits were presented.

In our specific implementation we translated the edge matching problem into a *combined placement* problem. This is, we implemented the edge matching technique using a combined placement approach based on simulated annealing. Given a placement of all the circuits separately in the reconfigurable region, a tunable circuit can easily be extracted. The LUTs positioned on the same physical LUT will be implemented using the same tunable LUT. The number of TCONs of this tunable circuit is used as a cost function during the combined placement. This is discussed in more detail in the following section.

4.2.1.1 Combined placement

A conventional FPGA placement algorithm takes two inputs: the mapped input circuit and a description of the target FPGA architecture. The algorithm searches a legal placement for the logic blocks of the input circuit so that circuit wiring is optimised. In a legal placement every LUT is associated to (placed on) one of the physical LUTs (without overlap).

As we explained in Section 2.2.3, the conventional placement tool is based on simulated annealing. The algorithm starts by randomly, but legally, placing the functional blocks in the input circuit on physical blocks of the FPGA architecture. Afterwards, the placer repeatedly tries to improve the placement cost by interchanging the functional blocks placed on two randomly chosen physical blocks. Such an interchange is called a swap.

The simulated annealing algorithm used in the combined placement step, of which the simplified pseudo code is shown in Figure 4.5, is very similar to that of the conventional placement. We extended the conventional placement tool to accommodate the simultaneous placement of several LUT circuits. First, the LUTs of all the circuits are placed randomly on the reconfigurable region. In the pseudo code we assume there are N circuits being placed. In the conventional placement only one LUT is allowed per physical LUT.

```
function combined_placement(Netlists NL_i, Architecture
    A):
  for i in 1 to N:
      Placement P_i = randomPlacement(NL_i, A)

  while (T > T_min):
      c = pickRandomCircuit()
      move = findLegalMove(P,c)
      ΔCost = costAfter(move, NL_i, P) − currentCost(
          NL_i, P)
      if (ΔCost < 0 or uniform(0, 1) < e^(−ΔCost/T)):
        P = acceptSwap(move, P)

  return P
```

Figure 4.5: Simplified pseudo code for the simulated annealing based combined placement algorithm.

In the case of the combined placement, however, LUTs belonging to different circuits can be placed on the same physical LUT.

During the combined placement algorithm, selecting a swap consists of two steps: selecting a circuit for which the swap will be executed and choosing two random physical blocks. Only the LUTs placed on the chosen physical LUTs belonging to the selected circuit will be interchanged, the LUTs of the other circuits maintain their position.

Just as in the conventional placement algorithm a swap is accepted if the cost function used reduces. Swaps that don't reduce the cost also have a chance of being accepted, which is dependent of the temperature T of the algorithm.

One of the advantages of using a simulated annealing based placement approach to place the circuits simultaneously is that we are able to easily experiment by adjusting the cost function used in the simulated annealing algorithm. In the case of edge matching the cost function consists of the number of TCONs of the tunable circuit being formed. During the combined placement algorithm a connection C is determined by its physical source and sink pin (so, si). Matching connections are counted only once. A collection of N con-

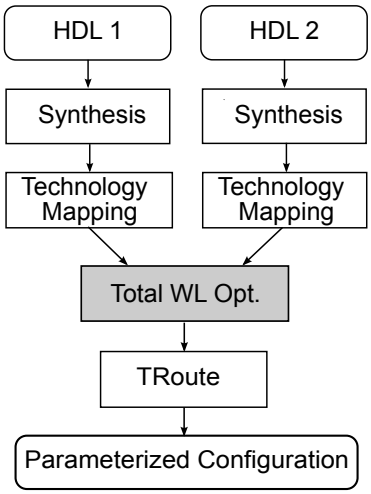


Figure 4.6: Tool flow that uses total wire length optimization of the tunable circuit when merging LUT circuits into a placed tunable circuit.

nections, $C_i = (so_i, si_i)$,with $i \in \{1, 2, ..., N\}$, *match* if $so_i = so_j$ and $si_i = si_j \; \forall i, j \in \{1, 2, ..., N\}$. We note that it is only useful to define matching of connections when the connections belong to different circuits.

4.2.2 Optimization of the total wire length

The combined placement algorithm in the previous section uses the number of TCONs of the tunable circuit being formed as a cost function. The benefit of using a simulated annealing approach is that different cost functions can be tried. Indeed, circuit edge matching only looks at the topology of the tunable circuit that is formed and does not take into account the placement of the tunable LUTs. However, using a combined placement strategy, the information regarding the placement of the LUTs also allows to asses an estimation of the wire usage of the tunable circuit being formed. To achieve this goal, the cost function used in the *total wire-length optimization* approach uses an estimation of the wire length TRoute will need to route the tunable circuit. The wire-length estimation used during the combined placement is the same as the one TPlace uses during the placement

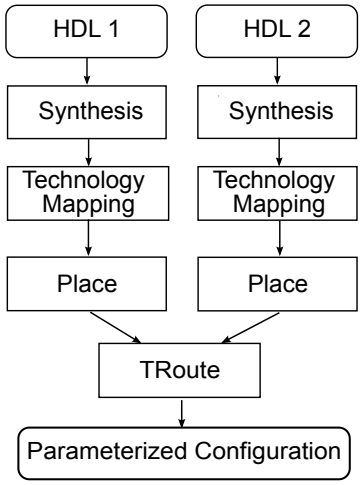


Figure 4.7: Tool flow that uses individual wire length optimization of the tunable circuit when merging LUT circuits into a placed tunable circuit.

of the tunable circuit after merging.

The tool flow proposed in this section is shown in Figure 4.6. Now the LUT circuits are merged into a tunable circuit during a combined placement step that optimizes the total wire length, generating a placed tunable circuit. Therefore only a routing step is necessary to complete the implementation.

4.2.3 Optimization of the wire length per circuit

The third and final tool flow also takes the wire length into consideration. The only difference with the tool flow proposed in the previous section is that the wire length is optimized per circuit. Figure 4.7 shows this tool flow. The main difference is that the placement step also happens separately. This flow also generates a placed tunable circuit which only needs to be routed. Note that the only step where a combined implementation occurs in this flow is the routing.

4.2.4 Experiments and results

4.2.4.1 Experimental set up

Benchmarks To validate the proposed tool flows we conducted experiments using 3 different types of applications: a regular expression matching application (RegExp), an adaptive filtering application (FIR) and general MCNC benchmarks. All experiments in this section implement 2 of these circuits in the reconfigurable region.

In [89] a tool was developed that can generate a hardware engine, written in VHDL, that matches a certain regular expression. In the first set of experiments, we chose 5 regular expressions out of the Bleeding Edge rules set [1] and with this tool generated the corresponding circuits. Then 10 experiments were executed using by picking all possible combinations of 2 circuits out of the 5 generated circuits. In the second set of experiments we combined 10 low pass and 10 high pass finite impulse response (FIR) filters. The non-zero coefficients were chosen randomly, after which all the constants were propagated. Such a FIR filter is 3 times smaller than the generic version.

Finally, in the third set of experiments, we chose 5 circuits out of the general MCNC benchmark suite [112] that were of similar size compared to the rest of the circuits in these experiments. Afterwards we executed 10 experiments by making all possible combinations of 2 circuits.

For every application the minimum, average and maximum number of LUTs are reported in Table 4.1.

Table 4.1: Size of the LUT circuits used in the experiments.

	Minimum	Average	Maximum
RegExp	224	243	261
FIR	235	302	371
MCNC	264	310	404

FPGA architecture The combined placement algorithm was implemented based on our Java version of the VPR (Versatile Place and Route) wire-length driven placer [13]. VPR is the most commonly

used academic tool for place and route algorithms. The FPGA architecture used for each of the implementations, is described in `4lut_sanitized.arch`. This is an FPGA architecture file included in the distribution of VPR. It has logic blocks containing one 4-LUT and one flip-flop and the wire segments in the interconnection network only span one logic block. We note that the techniques and tools we use are independent of the architecture used. The number of inputs of the LUTs is simply an input parameter of the tool flow. Also, different routing architectures can be used since TRoute uses a standard representation of the routing infrastructure, the routing resource graph [13].

Since there is no other functionality implemented on the FPGA, the reconfigurable region comprises the complete FPGA in our experiments. As recommended in [13], the square area of the FPGA and the channel width of this architecture were both chosen 20% bigger than the minimum needed.

Table 4.2: Results for the reconfiguration overhead (RO) (in bits)and wire length (WL) (in wires) for the conventional DPR flow.

Experiment	RegExp			FIR			MCNC		
	RO (in bits)	WL_1 (in wires)	WL_2	RO (in bits)	WL_1 (in wires)	WL_2	RO (in bits)	WL_1 (in wires)	WL_2
1	8944	1408	1674	10216	1854	1616	12493	3121	1899
2	9005	1408	1651	9689	1674	1394	9781	1880	990
3	8486	1439	1247	10159	1682	1685	10656	983	3024
4	8635	1392	1334	10090	1787	1597	11343	2156	1859
5	9114	1659	1637	10751	1717	1619	9029	2107	1024
6	8705	1652	1300	10382	1644	1382	11955	2038	2958
7	8897	1687	1381	10783	1632	1736	12524	2268	2736
8	8573	1643	1213	10781	1867	1589	12050	2839	1867
9	8807	1674	1338	11956	2242	1578	11005	2746	991
10	7789	1238	1347	11681	2195	1389	13466	2618	3130

Table 4.3: Results for the reduction of the reconfiguration overhead (RRO) (in %) and wire length increase (WLI) (in %) for the edge matching approach.

	RegExp			FIR			MCNC		
Experiment	RRO	WLI_1	WLI_2	RRO	WLI_1	WLI_2	RRO	WLI_1	WLI_2
1	34	84	56	36	71	82	41	38	118
2	35	75	70	34	74	84	33	47	132
3	33	51	82	35	88	84	42	186	28
4	34	51	73	35	76	70	37	43	106
5	38	48	42	33	80	72	38	35	152
6	36	49	59	33	71	90	42	97	48
7	36	45	60	33	101	75	40	59	52
8	35	52	75	34	72	78	39	39	104
9	36	48	66	34	60	97	39	31	192
10	35	65	60	36	42	86	42	97	47

(in %)

Table 4.4: Results for the reduction of the reconfiguration overhead (RRO) (in %) and wire length increase (WLI) (in %) for the total wire length optimization approach.

	RegExp			FIR			MCNC		
Experiment	RRO	WLI_1	WLI_2	RRO	WLI_1	WLI_2	RRO	WLI_1	WLI_2
1	35	26	19	35	17	24	40	21	31
2	35	27	20	33	34	35	32	27	60
3	33	19	36	33	36	33	41	38	11
4	33	29	36	33	24	28	35	26	39
5	38	23	24	34	23	21	36	13	29
6	34	25	24	33	17	24	41	28	23
7	36	23	34	33	28	27	40	28	25
8	33	23	35	33	12	31	38	19	41
9	35	18	33	35	14	32	37	9	34
10	34	25	24	34	22	36	42	28	21
				(in %)					

Table 4.5: Results for the reduction of the reconfiguration overhead (RRO) (in %) and wire length increase (WLI) (in %) for the wire length optimization per mode approach.

	RegExp			FIR			MCNC		
Experiment	RRO	WLI_1	WLI_2	RRO	WLI_1	WLI_2	RRO	WLI_1	WLI_2
1	34	12	8	35	9	13	41	9	15
2	36	13	7	33	10	11	32	10	10
3	33	4	9	33	8	15	41	14	5
4	33	12	13	35	10	25	37	10	17
5	36	15	8	35	6	8	37	4	9
6	34	7	7	35	14	16	41	17	14
7	36	7	4	31	15	10	40	8	9
8	35	4	11	33	14	12	38	5	22
9	35	8	23	36	3	12	37	4	16
10	34	8	11	35	9	13	42	16	9

(in %)

Table 4.6: Overview of the average reduction of reconfiguration overhead (RRO) (in %) and wire length increase (WLI) (in %) for the different approaches considered.

	RegExp		FIR		MCNC	
	RRO	WLI	RRO	WLI	RRO	WLI
Edge matching	35	60	34	78	39	83
Total wire length	34	26	34	26	38	27
Wire length per mode	35	10	34	12	39	11

(in %)

4.2.4.2 Results

We point out that both our approach and the conventional DPR flow have the same gains in area compared to a static implementation that does not use DPR. Instead of implementing the different circuits next to each other on the FPGA, only an FPGA region that can contain the biggest circuit is needed. Since both the conventional DPR approach and our new approaches time-multiplex the different circuits on the same FPGA region, they use the same amount of area. Two other metrics were used to further evaluate the quality of the implementation: reconfiguration overhead and wire length.

Table 4.2 presents the results obtained for the conventional DPR flow. Tables 4.3, 4.4 and 4.5 contain the results obtained for the edge matching flow, the total wire length optimization and the individual wire length optimization approach, respectively. These last tables were expressed in percentages relative to the conventional DPR flow. Table 4.6 summarizes the average results. It contains the average values for both the reduction of reconfiguration overhead (RRO) as the wire length increase (WLI), compared to the conventional DPR flow. These results are discussed in more detail below.

4.2.4.3 Reconfiguration overhead

In this section we assume the reconfiguration overhead to be directly proportional to the number of bits that needs to be rewritten in the configuration memory. The reconfiguration overhead gives an indi-

Table 4.7: Overview of the maximum and minimum reduction of the reconfiguration overhead (RRO) (in %) for the different approaches considered.

	RegExp		FIR		MCNC	
	Max	Min	Max	Min	Max	Min
Edge matching	38	33	36	33	42	33
Total wire length	38	33	35	33	42	32
Wire length per mode	36	33	36	31	42	32
			(in %)			

cation of how much time is needed to load the configuration in the configuration memory during run-time, i.e. the reconfiguration time. It is also a good indicator for the memory space needed to store the configuration bit stream.

The reconfiguration overhead is broken down into two parts: LUT bits and routing bits. In this work we assume that all LUT bits are rewritten. We do however count only the bits in the routing that are dynamic. A dynamic configuration bit has a different value for the different contexts or circuits. It therefore needs to be rewritten when one switches between the circuits implemented in the reconfigurable region. For our approach the dynamic bits of course correspond to the parameterized bits.

Note that it would be in the advantage of the combined implementation approach if the LUT bits would also be counted in a bit wise fashion, because this would increase the routing to LUT ratio. We choose however not to do this for two reasons. The techniques developed in this chapter do not optimize the correlation between LUT bits. The second reason is that this way the LUT bits do not need to be accessible per bit to apply this new technique.

In Table 4.6 we see that the tool flows proposed in this chapter all obtain a reduction of reconfiguration overhead of around 35%. The regular expression and adaptive filtering application obtain a reduction of around 34%, while a reduction of around 39% is achieved for the MCNC benchmarks. Table 4.7 shows the maximum and minimum values obtained for the reduction of reconfiguration overhead. We see that the results of the regular expression matching and the

adaptive filtering application are between 31% and 38%. The results for the MCNC benchmarks are between 32% and 42%.

In Figure 4.9 (a) the RRO of the three proposed tool flows over all benchmarks is shown in a whisker plot. The main observation here is that the edge matching approach, which was previously proposed in [82], on average does not further increase the correlation between configurations. The reduction of reconfiguration overhead mainly comes from the use of TRoute.

Figure 4.8 illustrates how the combined routing using TRoute reduces the number of bits that need to be rewritten in the configuration memory, compared to the conventional DPR flow. In Figure 4.8(a) a set of TCONs is shown that represents a simple 2:1 multiplexer we would like to implement using dynamic reconfiguration. The goal is to obtain two FPGA configurations. One corresponding to connection (i_0, o_0), the other one to connection (i_1, o_0). By reconfiguring the FPGA during run-time with the appropriate configuration we can choose which input gets connected with the output.

Figures 4.8(c) and (d) show a possible implementation of the connections (i_0, o_0) and (i_1, o_0) respectively, on a simple 2×1 FPGA, using the conventional DPR flow. Although the wire length of each of the implementations is optimal, we note that the switches being used in the two configurations are completely different. In this example the bits of all the 10 used switches would need to be changed to obtain an appropriate configuration. In Figure 4.8(b) we see the same example implemented using TRoute. TRoute stimulates sharing of routing resources between connections of different circuits. This results in a parameterized configuration with only 2 parameterized bits. Only these bits need to be rewritten to specialize the parameterized configuration into the appropriate regular configuration.

4.2.4.4 Wire length

In the previous section it was shown that a significant reduction of the reconfiguration overhead can be obtained using a combined optimization approach. However, this benefit does not come for free. In our proposed tool flows the different circuits are not implemented separately, as is the case in the conventional DPR flow, but instead a global solution is considered to generate an efficient tunable circuit. In this section we assess the impact this has on the wire length. Wire length is an important metric for the quality of a circuit, since it

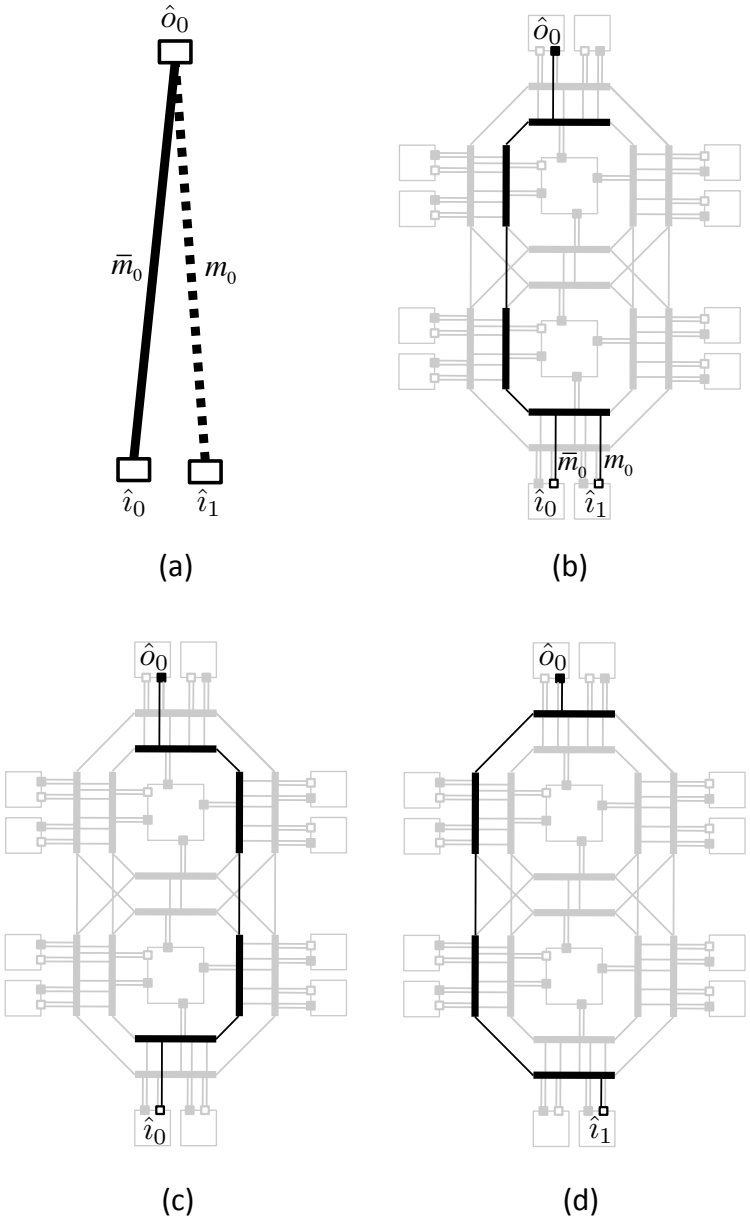


Figure 4.8: Schematic representation of a set of TCONs with the functionality of a 2:1 multiplexer (a). Combined implementation of that set of connections using parameterized configurations (b), compared to a separate implementation by the conventional DPR flow(c)(d).

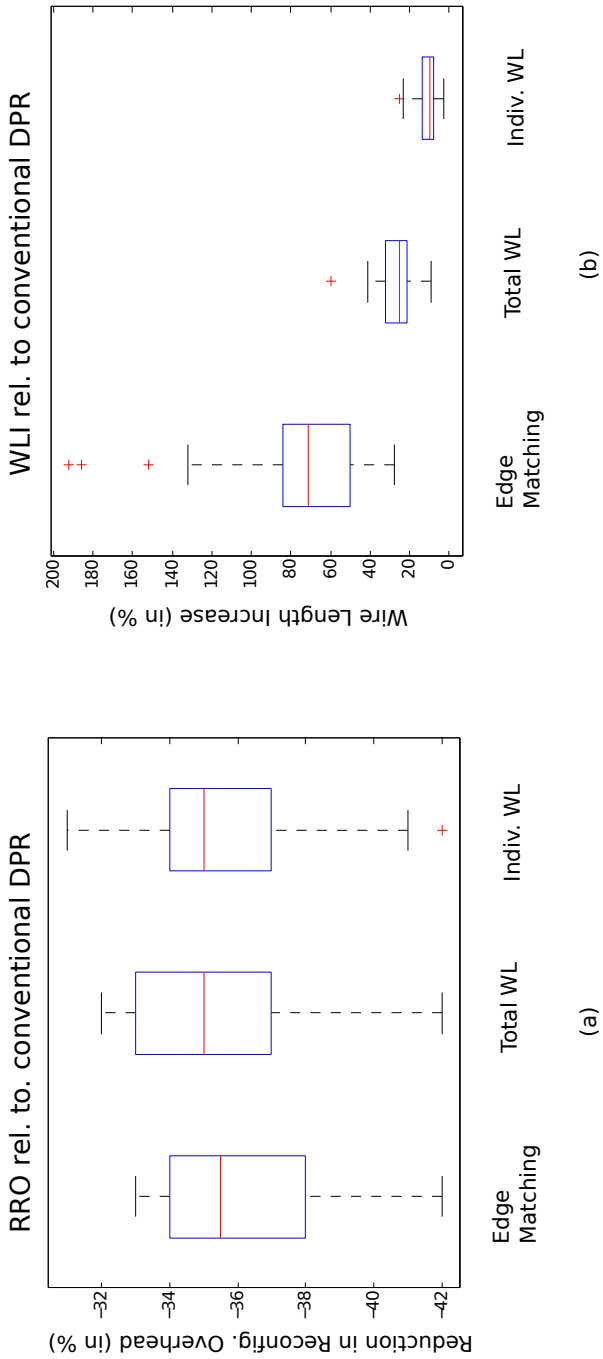


Figure 4.9: Whisker diagrams of the RRO (a) and WLI (b) of the explored tool flows (in % relative to conventional DPR.)

Table 4.8: Overview of the maximum and minimum wire length increase (WLI) (in %) for the different approaches considered.

	RegExp		FIR		MCNC	
	Max	Min	Max	Min	Max	Min
Edge matching	84	42	101	42	192	28
Total wire length	36	18	36	12	60	9
Wire length per mode	23	4	25	3	22	4
			(in %)			

correlates with power usage and performance (maximum clock frequency) of a circuit [13]. Each circuit uses a set of wires when it is active. We compare the size of this set in the case of implementation with conventional DPR and the three tool flows proposed in this chapter. Results are averaged over all experiments. We also present maximum and minimum values.

Table 4.6 contains the summary of the average results obtained in the experiments. Table 4.8 presents a summary of the maximum and minimum wire length increases.

In the previous section we saw that all the tool flows we proposed resulted in more or less the same reduction of reconfiguration overhead. Table 4.6 clearly shows that the edge matching approach increases the wire length considerably. The average wire length increase is 60%, 78% and 83% for the regular expression, adaptive filtering and MCNC benchmarks, respectively. In Table 4.8 we even see that for the MCNC benchmarks the maximum wire length increase is 192%. It seems that edge matching constrains the solution space that can be explored by the placement and routing algorithms. The wire length is best taken into consideration when generating the tunable circuit. Compared to the experiments done in [82] we also use bigger and more general applications. The work in [82] does not report results on impact of the quality of implementation of the circuits.

The total wire length optimization approach results in an increase in wire length of around 26% on average and maximum 60%. This is already much better than the edge matching approach. The approach that obtains the best results however is the individual wire length optimization approach. This approach results in an average increase

in wire length of around 11% and maximum 25%. Figure 4.9 (b) that plots the WLI of the three proposed tool flows over all benchmarks, visualizes the differences between the tool flows.

The importance of this wire length increase is dependent on the application. Note, however, that there are applications that do not run at their maximum performance, because system requirements are not that stringent. Since FPGAs are also used for parallel applications, such as regular expression matching, they often rely more on massive parallelism than on high clock frequencies for performance. There are therefore applications, for which the increase in wire length is not a major draw back, especially given the significant reduction of the reconfiguration overhead that can be obtained with this technique.

4.3 Combining more than 2 circuits

4.3.1 Experiments and results

4.3.1.1 Experimental set-up

The experimental set-up is the same as the one for 2 circuits in the previous section. Experiments were done using 3 types of benchmarks: a regular expression matching application, an adaptive filtering application and general MCNC benchmarks. In each experiment 10 implementations were generated with a combination of N circuits out of the benchmarks, where N was varied between 2 and 5. This way we can analyze the quality of the implementation as the number of circuits increases. For the adaptive filtering application no solution was found for N = 5. In this section we only consider the individual wire length optimization approach, since this had superior results in the previous section.

4.3.1.2 Results

4.3.1.3 Reconfiguration overhead

The average results for the reduction of the reconfiguration overhead (RRO) and wire length increase (WLI) are presented in Table 4.9. The maximum and minimum values for the reduction of the reconfiguration overhead are presented in Table 4.10.

Table 4.9: Overview of the average reduction of the reconfiguration overhead (RRO) (in %) and wire length increase (WLI) (in %), for increasing number of circuits.

	RegExp		FIR		MCNC	
Nr. of circuits	RRO	WLI	RRO	WLI	RRO	WLI
2	35	10	34	12	39	11
3	33	17	34	21	38	16
4	32	20	32	25	36	22
5	30	22			34	25

(in %)

We see in Table 4.9 that the reduction that can be obtained using the individual wire length optimization approach becomes slightly smaller as the number of circuits increases. It decreases by 5% for the regular expression matching applications and MCNC benchmarks. The reduction for the adaptive filtering application is 2%. The reduction of the reconfiguration overhead for 4 circuits is around 34% for all benchmarks. The minimum and maximum values more or less follow this trend.The reduction of reconfiguration overhead is between 29% and 36% for the regular expression matching and adaptive filtering applications. For the MCNC benchmarks this is between 32% and 42%.

4.3.1.4 Wire length

The average increase in wire length, as shown in Table 4.9, increases as the number of circuits is higher. For the regular expression matching and MCNC benchmarks this increases from around 11% for 2 circuits to around 24% for 5 circuits. For the adaptive filtering application it increases to 25% for 4 circuits. Table 4.11 shows the minimum and maximum values for the wire length increase. The minimum and maximum wire length increases are around 10 % and 30 %, respectively.

Table 4.10: Overview of the maximum and minimum reduction of reconfiguration overhead (RRO) (in %), for increasing number of circuits.

	RegExp		FIR		MCNC	
Nr. of circuits	Max	Min	Max	Min	Max	Min
2	36	33	36	31	42	32
3	35	32	35	33	41	34
4	33	30	33	31	38	33
5	32	29			37	32

(in %)

Table 4.11: Overview of the maximum and minimum wire length increase (WLI) (in %), for increasing number of circuits.

	RegExp		FIR		MCNC	
Nr. of circuits	Max	Min	Max	Min	Max	Min
2	23	4	25	3	22	4
3	29	11	38	13	27	4
4	30	11	32	18	31	7
5	33	14	-	-	35	15

(in %)

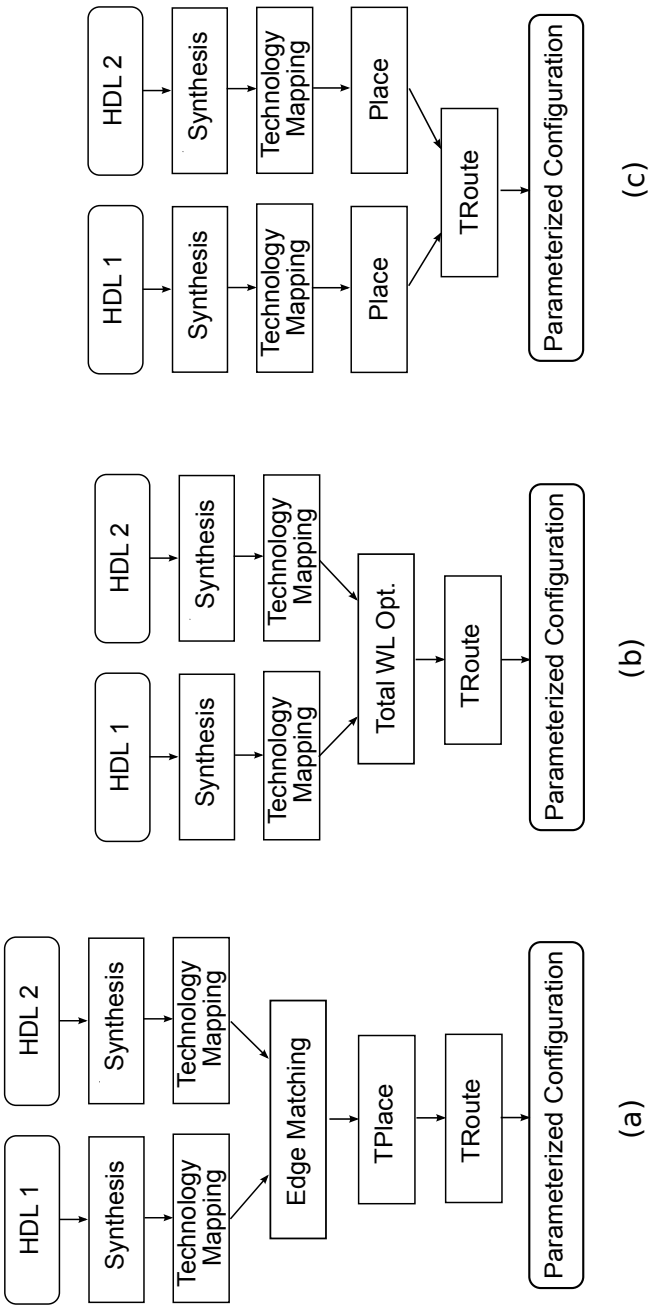


Figure 4.10: Overview of the three tool flows developed in this chapter: edge matching (a), total wire length optimization (b) and individual wire length optimization (c).

4.4 Discussion

Dynamic reconfiguration allows to implement several circuits in the same reconfigurable region. This can increase the area efficiency of the implementation significantly. The downside of this approach, however, is the reconfiguration overhead. To switch between circuits the configurations need to be loaded into the configuration memory, which takes considerable time. The configurations also take up memory space.

The conventional DPR flow implements the circuits separately in the reconfigurable region without taking the reconfiguration overhead into account. It does this independent of the number of circuits being implemented. In contrast, in this chapter we explored three novel tool flows in which a limited number of circuits are implemented taking the reconfiguration overhead into account. Because most bits of the configuration memory control the routing, the focus is on reducing the reconfiguration overhead of the routing. This is done by using a combined implementation approach at a certain stage of the tool flow. The three tool flows are all shown in Figure 4.10, to allow side-by-side comparison.

The first tool flow uses edge matching to generate a tunable circuit that has a reduced number of connections. This tunable circuit is then further implemented using the TPlace and TRoute tools of the DCS tool flow. The second and third tool flow generate a tunable circuit that is already placed and only needs to be routed using TRoute. The second tool flow uses a cost function during a combined placement step that takes into account the total wire length of the tunable circuit. The third tool flow optimizes the wire lengths of the different circuits separately during placement.

From the experiments I conducted in this chapter it turns out that the reduction of reconfiguration overhead that can be obtained using a combined implementation approach is around 35%. This reduction of reconfiguration overhead comes from the combined routing approach used by TRoute. TRoute optimizes the total wire length of the tunable circuit and as such reduces the number of dynamic bits in the routing. This reduction of reconfiguration overhead does not come for free, it is associated with an increase in wire length of the implemented circuits. The edge matching approach, previously proposed in [82], does not increase the reduction of reconfiguration overhead obtained using TRoute. It is, however, associated with a

high increase in wire length. Compared to the experiments done in [82] we use bigger, more general circuits and report the impact on the quality of implementation of the circuits.

The third tool flow obtained the best results, with an increase in wire length of, on average, around 11% for 2 circuits. This tool flow was further researched for increasing number of circuits. The experiments in this chapter show that as the number of circuits increases, the reduction of configuration data lowers with 5%. The increase in wire length is more noticeable, it increases from around 11% for 2 circuits to around 23% for 4 circuits on average. No solution was found for the adaptive filtering application for 5 circuits. The combined optimization tool flows could not find a solution for all the applications considered when the number of circuits was more than 5.

In this chapter the reconfiguration overhead is expressed as the number of bits that needs to be rewritten in the configuration memory. For the routing we counted the number of dynamic bits. However, the configuration memory of current commercial FPGAs is organized in frames. A configuration frame needs to be rewritten completely, even when it contains only one dynamic bit. Although much lessons are learned from the techniques developed in this chapter, these would only be useful if the routing's configuration memory would be bit-accessible. In the next chapter other techniques are developed that can be used directly in a frame-based reconfiguration approach. These techniques were developed based on lessons learned in this chapter.

Chapter 5

Reducing the overhead of limited-context DPR in a frame-based reconfiguration approach

As was mentioned in the introduction, the configuration memory of FPGAs consists of SRAM memory cells that control the contents of the lookup tables and the state of the routing switches. To implement a circuit on the FPGA, a configuration needs to be generated. This configuration contains the binary values that need to be written to the FPGA's configuration memory cells. In conventional DPR systems, a configuration is generated for every circuit by implementing it independently in the reconfigurable region (RR). Each memory cell of the RR then corresponds to one binary value for each circuit. When these binary values are the same for a single memory cell in all circuits, they are called a static bit. Otherwise, they are called a dynamic bit. Memory cells containing a static bit do not need to be rewritten when switching between circuits.

However, in current FPGAs, the reconfiguration granularity is a collection of memory cells called a frame. A whole frame needs to be rewritten, even when only one memory cell of the frame contains a dynamic bit. As we will show further on, the problem with conventional DPR systems is that the different circuits are implemented separately in the reconfigurable region. The dynamic bits are therefore scattered over the frames of the configuration memory. This makes it

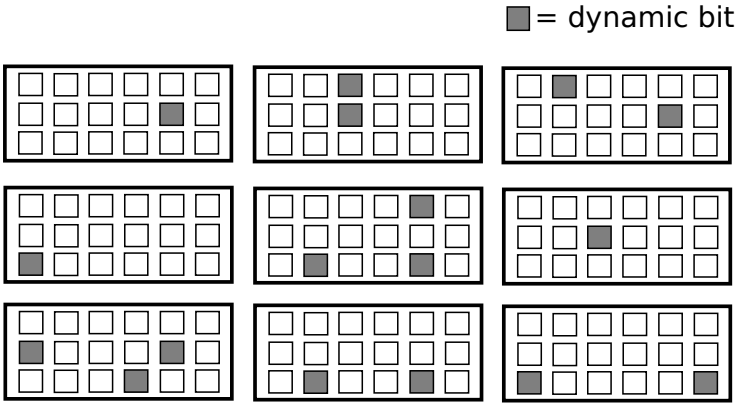


Figure 5.1: Scattering of the dynamic bits when using the conventional DPR tool flow.

necessary to reconfigure almost the complete reconfigurable region. This is illustrated in Figure 5.1, for a simplified configuration memory where each frame contains 18 bits. Our goal is to cluster the dynamic bits so that less frames need to be reconfigured.

An FPGA configuration consists mostly of routing bits. Figure 5.2 shows, for example, that in the experiments of this chapter only 30% of the configuration memory is used for the Configurable Logic Blocks (CLBs). The rest is used for the configuration of the switch blocks (SBs) and connection blocks (CBs) in the routing. This is the main reason that in this chapter we focus on the routing algorithm.

In this chapter, I present two routing algorithms, called StaticRoute and ClusterRoute, that can cluster the dynamic bits in fewer frames, and thus increase the number of routing frames without dynamic bits. This reduces the reconfiguration overhead, because the routing frames without dynamic bits are constant for all the implemented circuits and thus do not need reconfiguration during runtime.

5.1 StaticRoute: clustering dynamic bits in dynamic regions

The approach taken in this section consists of two steps. In a first step the configuration memory of the RR's routing switches is di-

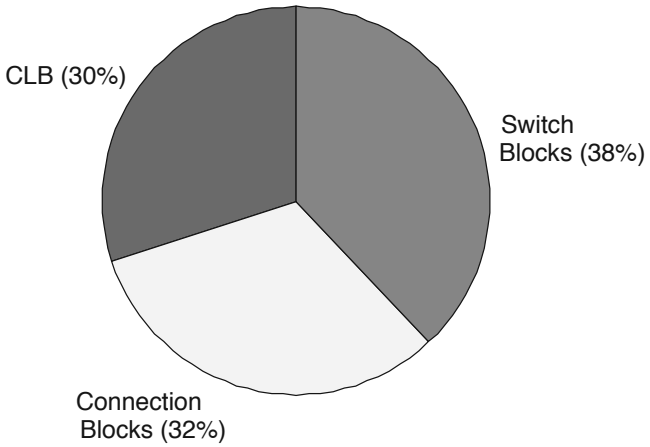


Figure 5.2: The relative contribution of CLBs and routing to the size of the configuration memory.

vided into a static and a dynamic part. Care needs to be taken that the memory cells of the static part reside in other frames than those of the dynamic part. This way the configuration frames are split up in static and dynamic frames. Then, in a second step the interconnections of *all* circuits are routed simultaneously using a novel router, which is called StaticRoute. StaticRoute routes the interconnections in such a way that dynamic bits are avoided in the static switches of the RR. The dynamic bits are thus clustered in the dynamic part of the configuration memory. This is illustrated in Figure 5.3, where the static frames (marked with S) do not contain dynamic bits. To the best of my knowledge, I was the first to propose such an approach.

StaticRoute uses a novel concept called *switch congestion*. In contrast to wire congestion, where a wire is congested if multiple connections try to use it, a switch is said to be congested when it is in a static part, but is controlled by a dynamic bit. In both cases the congestion represents situations which need to be avoided in the final solution. StaticRoute is based on the PATHFINDER algorithm, explained in Section 2.2.4, and also uses the negotiated congestion mechanism to resolve both wire and switch congestion.

To implement the circuits on the FPGA, a router alone is not enough, but a complete tool flow is needed. In Section 5.1.1, the StaticRoute tool flow is presented and compared to the conventional

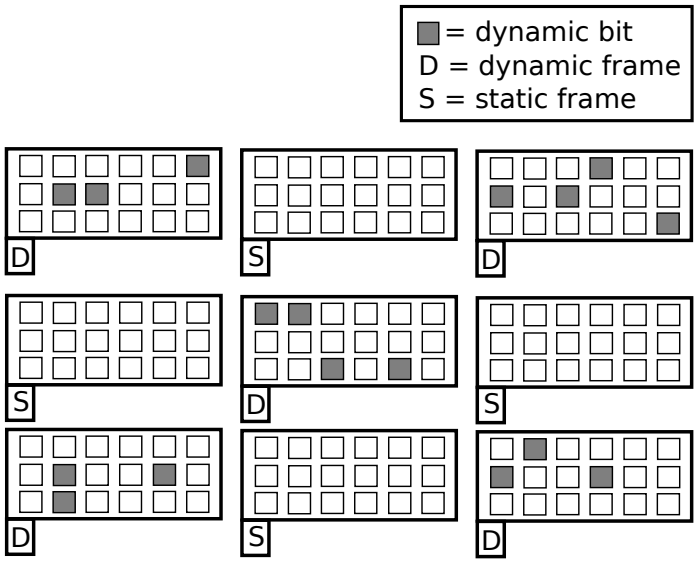


Figure 5.3: Clustering of the dynamic bits in dynamic frames when using the StaticRoute tool flow.

DPR tool flow.

Before StaticRoute is used the routing switches need to be marked as being either static or dynamic. We note that this happens in the CAD software. To be able to do this, the representation of the FPGA's architecture needs to be extended. This is explained in Section 5.1.2.

Detecting dynamic bits after the configurations are generated is easy. When a memory cell has different values in the different configurations, it contains a dynamic bit. This means it will have to be rewritten during run-time. In Section 5.1.3 we show, however, that it is also possible to detect dynamic bits during routing.

Section 5.1.4 handles how StaticRoute extends the cost function of PATHFINDER, so that dynamic bits are avoided in the static part of the configuration memory.

Section 5.1.5 and Section 5.1.6 present the results of thorough experiments done on a 6-LUT architecture based on the commercially available Stratix IV FPGA. Section 5.1.5 explores how the static part is best chosen in the FPGA's configuration memory. Our approach is specifically focused on the case where the number of circuits to be implemented in the reconfigurable region is limited. In Section 5.1.6

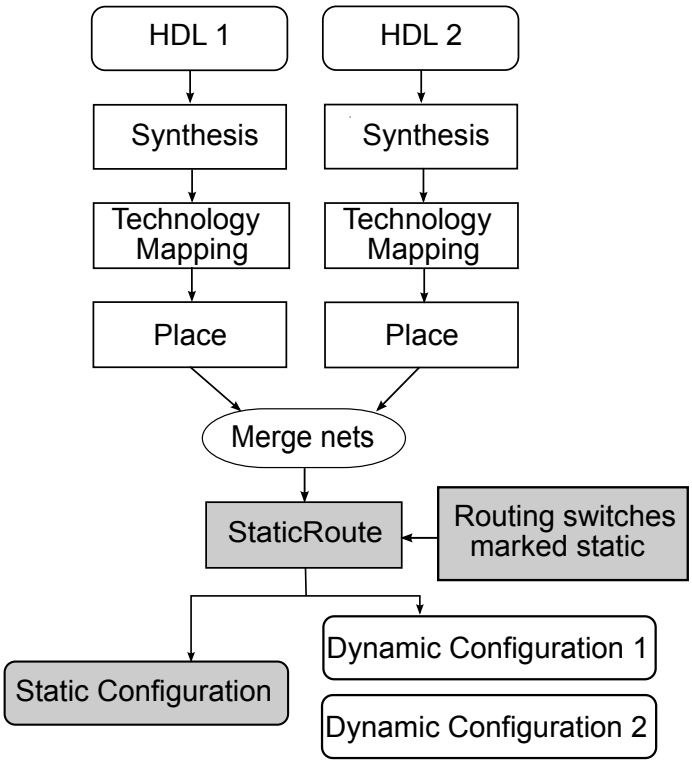


Figure 5.4: The StaticRoute tool flow. StaticRoute also takes as input the set of switches that are marked static.

the impact on performance is discussed as the number of circuits increases.

5.1.1 The StaticRoute tool flow

The proposed tool flow is presented in Figure 5.4. Instead of running the tool flow completely separately for the different circuits, the idea is to have a combined routing of the circuits. In this case, the tool flow is run separately until placement, generating a placed design for each circuit. Then the nets of all the circuits are merged into one set of nets. The tool flow was designed this way because from Chapter 4 it followed that a combined placement algorithm tends to increase the wire length of the circuits significantly and it therefore was better to focus solely on the routing algorithm [5].

The set of merged connections is routed with StaticRoute. StaticRoute thus routes the nets of *all* circuits simultaneously. This is of course only viable when the number of circuits is limited. StaticRoute detects dynamic bits and clusters them in the dynamic routing frames. Figure 5.4 shows that the result of the StaticRoute flow is one static configuration and a dynamic configuration for every circuit. The static configuration contains the binary values for the static part of the RR's routing. It only needs to be loaded to the FPGA's configuration memory once at start-up. The dynamic configurations contain the remainder of the configuration of the RR, needed to reconfigure the Configurable Logic Blocks (CLBs) and the dynamic part of the routing. These are used to switch between circuits during run-time. Since the dynamic configurations are much smaller than a configuration of the complete RR, the reconfiguration overhead can be reduced considerably.

5.1.2 Extended routing resource graph

As discussed in Section 2.2.4, in a standard routing resource graph (RRG) the nodes represent wires and the directed edges represent switches. In conventional routing algorithms, information needs to be associated only with the wires. However, when developing the StaticRoute routing algorithm for DPR, a new representation is necessary to associate information with the switches during routing. This can be a cost or information on which circuits are using a certain switch. This information can then be used to associate a switch congestion cost with the switches. Therefore, the routers in this chapter do not make use of a standard RRG, but of an extended RRG. This extended RRG does not only represent the wires as nodes but also the switches. An example of an extended RRG is shown in Figure 5.5. The round nodes are wires and the square nodes switches.

We note that this is a straight-forward extension of the representation of the routing architecture. Also adding information to the switches during routing happens in the CAD software and not in the actual architecture. The FPGA and its routing architecture do not need any adaptation for the routing algorithms described in this chapter.

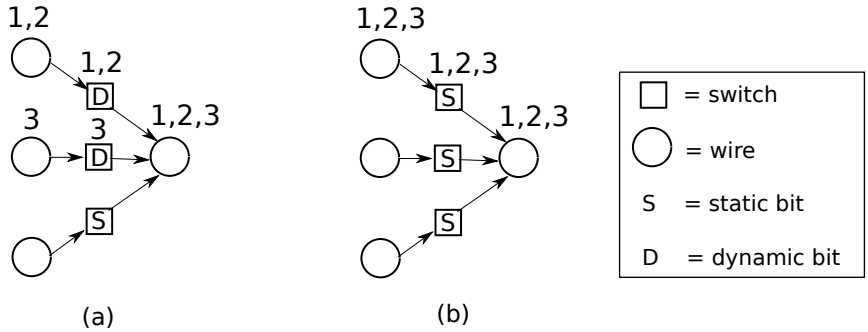


Figure 5.5: An example of an extended RRG of a multiplexer controlled by some dynamic bits (a) and one controlled only by static bits (b). The round nodes are wires and the square nodes switches.

5.1.3 Detecting dynamic bits in the extended routing resource graph

As can be seen in Figure 5.4, the input to StaticRoute is the list of the nets of *all* circuits that need to be implemented in the reconfigurable region. These nets are all annotated with the ID of the circuit they belong to. When a net uses a node during routing, it is also annotated with this ID. So, each node stores information regarding the updated set of circuits that use that node. Our starting point in this section is therefore an extended RRG annotated with this information.

As will be explained later, to take the reconfiguration overhead into consideration, also the switches will be associated with a cost. That is why we need to detect dynamic bits, to be able to calculate that cost. In this subsection we only discuss the detection of whether a switch contains a static or dynamic bit. The novel cost function, that combines the detection of dynamic bits with the information about how the switches are organized in frames, is discussed in Section 5.1.4.

Let us assume 4 circuits, numbered 1 to 4, are implemented in the RR. In Figure 5.5(a) we see a routing multiplexer of the RR, represented as an extended RRG. It connects the top wire to its output for circuits 1 and 2. The middle wire is connected to the output for circuit 3. Circuit 4 does not use the routing multiplexer and therefore is not shown in the figure. Let us focus on the top switch. It follows that this switch needs to be closed for circuit 1 and 2. It needs to be

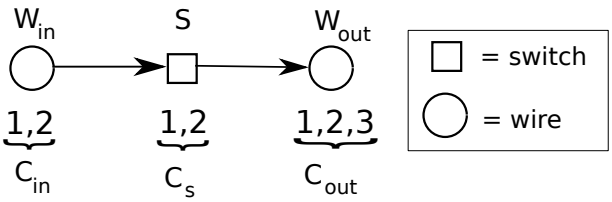


Figure 5.6: An example of a switch S controlled by a dynamic bit.

open for circuit 3, as not to add any extra capacitance of the wires of the other circuits. It has a don't-care value for circuit 4, because this circuit is not using this multiplexer. In this case, the switch clearly is controlled by a dynamic bit, since it has different values for different circuits.

Let us look at a second example in Figure 5.5(b). In this case the top switch has value 1 for circuits 1, 2 and 3. And it has a don't-care value for circuit 4. It is clear that when a switch and its connected wires are used by the same circuits, it does not have to be changed during run-time. The switch is closed for the circuits that use it and has a don't-care value set to 1 for the other circuits. The remaining switches are static, because they are not used by any circuit. They always have the static value zero.

In general, in the extended RRG, a switch node S connects two wire nodes W_{in} and W_{out}. Let us assume that S is used by a set of circuits C_S. W_{in} and W_{out} are used by C_{in} and C_{out} respectively. We state that S is controlled by a dynamic bit if:

$$((C_S \neq C_{in}) \vee (C_S \neq C_{out})) \wedge C_S \neq \phi. \tag{5.1}$$

Figure 5.6 shows an example with all the terminology indicated on the figure. In this figure we see that both switch S and wire node W_{in} are used by circuits 1 and 2, whereas the wire node W_{out} is used by circuits 1, 2 and 3. Switch S therefore contains a dynamic bit as C_S is different from C_{out}.

There is, however, a property that allows to express this condition differently: if a circuit uses a switch, then this circuit will also use both connected wires. The condition to detect a dynamic bit in switch S is therefore also expressed as:

$$((|C_S| < |C_{in}|) \vee (|C_S| < |C_{out}|)) \wedge |C_S| \neq 0. \tag{5.2}$$

The conditions $C_S \neq \phi$ and $|C_S| \neq 0$ are necessary to exclude unused switches, which are always static. We note that Equation 5.1 and Equation 5.2 are equivalent, they do not affect the end result. Both can be used to detect dynamic bits. Therefore for both conditions exactly the same results in terms of reduction of reconfiguration overhead and maximum clock frequency are obtained. In the implementation of this chapter we used Equation 5.2.

5.1.4 Cost function

In the previous sections we explained that in the extended RRG some switches are marked as being static. We also presented a way to detect dynamic bits in the extended RRG. In this section we introduce the term *switch congestion*. A switch is said to be congested when it is marked as static, but is controlled by a dynamic bit.

In the PATHFINDER algorithm the cost of using a wire only takes into account wire congestion. The nets are ripped up and rerouted until there are no wires that are congested. In this section we describe how we extended this algorithm to also take switch congestion into consideration. StaticRoute rips up and reroutes the nets of all circuits until all wire *and* switch congestion is resolved.

In the PATHFINDER algorithm a connection of a net is routed by searching the path of wire nodes with lowest cost in the RRG. In our algorithm the same happens in the extended RRG. Except that, to take switch congestion into consideration an extra cost per wire is added. Out of Equation 5.2 follows that when a wire is used, the congestion of all the static switches that are connected with it in the extended RRG are affected. That is why the cost of switch nodes in the path of the RRG is zero. So, only the wires contribute to the cost of a net. Switch nodes are used to hold information needed to determine the switch congestion penalty.

In our novel cost function, the cost of a node in the extended RRG is

$$cost(n,c) = \begin{cases} cost_w(n,c) + cost_s(n) & \text{if } n \text{ is a wire} \\ 0 & \text{if } n \text{ is a switch} \end{cases} \tag{5.3}$$

where $cost_w(n,c)$ is the wire congestion cost associated with wire node n and circuit c (we discuss the influence of c later). The term $cost_s(n)$ is the switch congestion cost associated with wire node n.

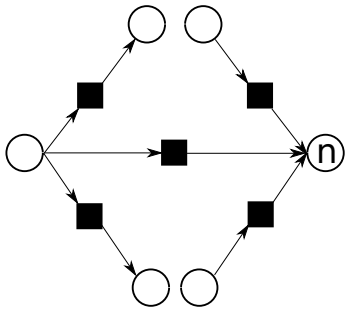


Figure 5.7: Example where the switches of the set S(n) for a wire node n are indicated in black in the extended RRG.

The term $cost_w(n, c)$ takes wire congestion into consideration and is very similar to Equation 2.1. Remember that StaticRoute routes the nets of *all* circuits simultaneously. However, when a net of the circuit c is routed, only the other nets of c are taken into consideration for the wire congestion. This is because nets of the other circuits do not cause wire congestion. They are never present on the FPGA at the same time as circuit c and therefore can share wires with circuit c. The equation for $cost_w(n, c)$ is

$$cost_w(n, c) = p(n, c) \cdot h(n, c), \tag{5.4}$$

where $p(n, c)$ and $h(n, c)$ are the present and history wire congestion penalty for circuit c. These are calculated as in Equation 2.2 and 2.3, the only difference is that $o(n)$ is replaced with $o(n, c)$.

The term $cost_s(n)$ takes switch congestion into consideration. As mentioned in the previous section, to determine whether a switch is congested the information associated with both the wire nodes it connects is needed. However, when a wire node is being evaluated during routing, it is not yet known what the next wire node will be. Therefore the calculation of the switch congestion cost of the switches in the fan-out nodes of a wire node is delayed until the next wire node is reached. Therefore the router calculates the cost for switch congestion using the union of the fan-in switch nodes of the current wire node n and the fan-out switch nodes of the previous wire node in the routing path currently being evaluated. This set of switch nodes is called $S(n)$. Figure 5.7 shows an example where the switches of $S(n)$, associated with a wire node n, are identified in

black. In this set $S(n)$, we use Equation 5.2 to identify the subset of congested switches, which we call $C(n)$.

Given a wire node n, with its associated set of congested switches $C(n)$, we propose the following equation for the switch congestion penalty

$$cost_s(n) = p_s(n) \cdot h_s(n), \tag{5.5}$$

where $p_s(n)$ and $h_s(n)$ are the present and history switch congestion penalty. The factor $p_s(n)$ resolves switch congestion during one iteration and is given by:

$$p_s(n) = 1 + |C(n)| \cdot p_f. \tag{5.6}$$

Note that if the use of a wire results in more congested switches this wire is more penalized. This is similar to the PATHFINDER algorithm, in which wires with more overuse result in a higher penalty.

The factor $h_s(n)$ takes into consideration the switch congestion that occurred in the previous iterations. It uses the congestion map that is built in the switch nodes. It is given by:

$$h_s(n) = \sum_{m \in C(n)} h_s(m), \tag{5.7}$$

where $h_s(m)$ is the history switch congestion penalty of one switch node m. This is updated every routing iteration i as follows:

$$h_s^i(m) = \begin{cases} 0 & \text{if } i = 1 \\ h^{(i-1)}(m) & \text{if } m \text{ is not congested} \\ h^{(i-1)}(m) + h_f & \text{otherwise} \end{cases} . \tag{5.8}$$

5.1.5 Selection of the static part

In this section I explore how the static part in the configuration memory of the routing is best selected. I will do this exploration by looking at two important quality metrics of the implementation: reconfiguration overhead and maximum clock frequency of the circuits. The reconfiguration overhead gives an indication of the efficiency of the reconfiguration process. The maximum clock frequency is important for the performance during the operation of a circuit.

I point out that both the StaticRoute tool flow and the conventional DPR flow have the same gains in area compared to a static implementation of the circuits on the FPGA. Instead of implementing the different circuits next to each other on the FPGA, only an FPGA region that can contain the biggest circuit is needed.

5.1.5.1 Experimental set-up

Integration in the VTR framework The novel tool flow, depicted in Figure 5.4, was implemented in the latest version of the VTR (Verilog-To-Routing) framework [80], which incorporates the latest 6.0 version of VPR (Versatile Place and Route). This extensive framework is written in C and consists of thousands of lines of code. Within this framework, StaticRoute was based on the wire-length driven router of VPR 6.0 [13] [66]. Wire-length is an important metric for the quality of the implementation of a circuit.

In the section above I described a new routing methodology for DPR of FPGAs, which could be integrated in any FPGA router that is based on the PATHFINDER algorithm. I chose to build in the StaticRoute tool flow into the VTR framework, because this has several advantages. First, the built-in verification algorithms can be re-used for our tool flow. VTR for example checks that a routing describes a properly connected tree for each net and that this tree connects all the pins used by that net. It also checks that no routing resources are overused (the occupancy of everything is recomputed from scratch) [80]. Second, VTR has a built-in timing analyzer and the provided architecture file contains timing information. Therefore it is possible to get representative timing information. For each implementation of a circuit it is possible to get the maximum clock frequency. Finally, because the FPGA architecture is represented in a standard XML format, it is possible to try out different architectures more easily.

I chose to do the experiments in VTR primarily because a built-in timing analyser is available. An additional benefit of the VTR framework is that an architecture file is available, which is strongly based on a commercially available FPGA, namely the Altera Stratix IV FPGA [9]. This FPGA architecture is used to conduct the experiments in this chapter.

Another alternative, the Rapidsmith framework, makes it possible to implement the circuits on a commercial (Xilinx) FPGA, but in

this framework timing information is not provided [60]. It would therefore be impossible to investigate the impact of our technique on the maximum clock frequency of the circuits.

FPGA architecture The architecture based on the Altera Stratix IV is described in `sample_arch.xml` and included in the distribution of VTR. This is the architecture used in the experiments of this chapter. It has configurable logic blocks (CLBs) containing 10 6-LUTs and the wire segments in the interconnection network span four logic blocks. It uses Wilton switch blocks [70] and unidirectional wires [61]. Altera's Chip Planner Tool was used to determine the channel width of the Altera Stratix IV [8], which is 248. This way the experiments are carried out as close as possible to the actual FPGA chip.

Note that the techniques and tools used in this chapter are independent of the FPGA architecture used. The number of inputs of the LUTs is simply an input parameter of the tool flow. Also different routing architectures can be used since StaticRoute uses a straightforward extension of a standard representation of the routing infrastructure called the routing resource graph. Because of this it would also be possible to target Xilinx FPGAs.

Since there is no other functionality implemented on the FPGA, the reconfigurable region comprises the complete FPGA in our experiments. The minimum square area of the FPGA was chosen to fit all circuits under consideration.

Choice of static configuration frames Unfortunately, detailed information on how the frames in the configuration memory are built up is not provided by the FPGA manufacturers. This is considered proprietary information and is not disclosed. How the configuration memory is organized in frames also differs between FPGAs. Given the limitations of physical design of the FPGA, it is however likely that internal structures which are close together in the FPGA architecture, will be controlled by memory cells which are close together in the configuration memory. For the sake of the experiments, it is therefore assumed that the frames coincide with the connection blocks (CBs) and the switch blocks (SBs). The size of an SB frame in the VTR architecture contains around 1300 bits which is approximately the same as the frame size for a commercially available FPGA, for example the Virtex V FPGA from Xilinx which has a frame size of 1312 bits [107]. The size of a CB frame in our architecture is of the

same order of magnitude, but considerably smaller, namely around 600 bits. An important reason to choose the frames as such, is that this makes it possible to research the interaction between these fundamental routing structures and the routing algorithms described in this chapter. It can among others be investigated which of these structures is most suitable to make static or is better kept dynamic.

To investigate the impact of marking routing blocks static, the cases are compared where the fraction of both the static SBs and static CBs varies between 0% and 75% in steps of 25%. This gives us 16 cases organized in a two-dimensional table as can be seen in Table 5.5. In doing so, it can be examined how these structures are best marked static. Figure 5.8 shows an example where 50% of the switch blocks are marked static. As can be seen, this is done in such a way that the selected blocks are spread uniformly over the FPGA's area. The cases where either 100% of the connection blocks or switch blocks are marked static are omitted from the table because StaticRoute was not able to find a solution for these cases. This already indicates that some dynamic flexibility is needed in both the switch and connection blocks.

Benchmarks For each of the cases above, that represents a relative portion of the CBs and SBs marked static, 20 experiments were conducted. In each of these experiments 2 circuits were randomly chosen out of the 20 largest circuits of the MCNC benchmark suite [112]. The circuits were chosen randomly to demonstrate that our novel tool flow can be used maintaining the possibility to implement completely unrelated circuits in the DPR region. These benchmarks can also be found in the 'benchmarks/blif' folder of the VTR framework. Their size varies from around 1000 6-LUTs to around 5000 6-LUTs. The chosen circuits for each experiment are listed in Table 5.1 and Table 5.2.

5.1.5.2 Results

Impact on reconfiguration overhead In both the case of conventional DPR as well as the StaticRoute tool flow, the configurations are computed off-line. This is done by running the tool flows shown in Figure 3.2 and Figure 5.4, respectively. During run-time the configurations only need to be downloaded to the FPGA's configuration memory when it is needed to switch between circuits.

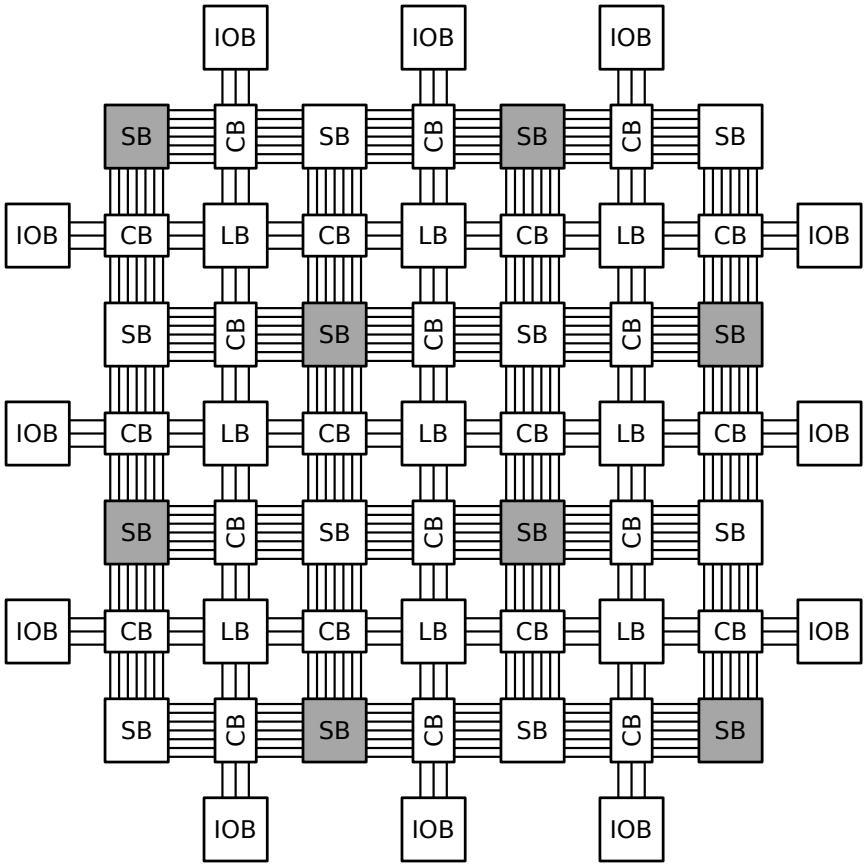


Figure 5.8: An example of a 3×3 island style FPGA where 50% of the switch blocks are marked static (in grey). LB is (Configurable) Logic Block, SB is Switch Block, CB is Connection Block and IOB is Input-Output Block.

Table 5.1: Names of the circuits used in the experiments 0 to 9.

Exp.	N = 2		N = 3			N = 4			
	Circ. 0	Circ.1	Circ. 0	Circ. 1	Circ.2	Circ. 0	Circ. 1	Circ. 2	Circ.3
0	s298	ex5p	tseng	frisc	dsip	elliptic	apex2	diffeq	misex3
1	s38584.1	ex1010	apex4	s38584.1	seq	misex3	clma	alu4	dsip
2	misex3	s38584.1	spla	s298	apex2	diffeq	ex1010	bigkey	elliptic
3	frisc	dsip	ex5p	seq	ex1010	bigkey	seq	frisc	apex4
4	diffeq	bigkey	apex2	dsip	misex3	des	frisc	clma	elliptic
5	elliptic	ex5p	s38417	apex2	apex4	diffeq	tseng	s298	elliptic
6	s38417	diffeq	ex5p	s298	frisc	s298	frisc	apex4	ex5p
7	alu4	ex5p	seq	spla	tseng	s38417	tseng	des	s298
8	clma	tseng	des	s38417	bigkey	s38417	s38584.1	ex5p	dsip
9	apex4	ex1010	bigkey	elliptic	clma	apex4	des	alu4	tseng

Table 5.2: Names of the circuits used in the experiments 10 to 19.

Exp.	N = 2		N = 3			N = 4			
	Circ. 0	Circ.1	Circ. 0	Circ. 1	Circ.2	Circ. 0	Circ. 1	Circ. 2	Circ.3
10	clma	des	s38417	s38584.1	elliptic	dsip	apex2	s38584.1	seq
11	bigkey	alu4	frisc	s38584.1	dsip	bigkey	alu4	dsip	tseng
12	alu4	seq	ex5p	s38417	s298	pdc	s38584.1	s38417	des
13	spla	tseng	frisc	pdc	s38417	apex4	seq	dsip	des
14	diffeq	s38584.1	clma	tseng	misex3	ex5p	dsip	misex3	clma
15	pdc	apex2	s38584.1	pdc	dsip	clma	alu4	apex2	s298
16	des	apex2	misex3	elliptic	bigkey	alu4	s38417	ex5p	des
17	s298	seq	apex4	frisc	des	clma	diffeq	ex1010	ex5p
18	dsip	spla	ex5p	des	spla	seq	alu4	pdc	tseng
19	pdc	misex3	bigkey	s298	dsip	diffeq	s298	elliptic	spla

The reconfiguration overhead (RO) is assumed to be proportional to the size of the configuration needed to reprogram the reconfigurable region. Therefore, to obtain the reduction of reconfiguration overhead (RRO) in the experiments the size of the configurations of the reconfigurable region is calculated, for both the conventional DPR as for the StaticRoute case and these are compared.

In the case of conventional DPR both the routing as well as the configurable logic blocks (CLBs) in the reconfigurable region (RR) are reconfigured. The size of the configuration needed for this is therefore given by:

$$B_{conv} = B_{conv}^{CLB} + B_{conv}^{R} \tag{5.9}$$

where B_{conv}^{CLB} and B_{conv}^{R} are the number of configuration bits in the reconfigurable region that control the CLBs and the routing, respectively.

$$B_{conv}^{CLB} = B_{total}^{CLB} = L_{CLB} * 64 + \sum_{m \in M_{CLB}} C_m \tag{5.10}$$

where L_{CLB} is the total number of LUTs contained in the CLBs of the RR, M_{CLB} is the set of multiplexers found in the cross bar switches of the CLBs and C_m is the number of configuration bits needed to control one multiplexer. The number of LUTs is multiplied by 64 as our architecture contains LUTs with 6 inputs. Note that for the StaticRoute tool flow it does not matter how the CLB bits are divided into frames, as these are always completely overwritten during runtime. C_m is given by

$$C_m = 2 * \left\lceil \sqrt{I_m} \right\rceil \tag{5.11}$$

where I_m is the number of inputs of the routing multiplexer. The upper bound takes non-square numbers of inputs into consideration. This formula is used because all multiplexers are considered to be two-level and one-hot encoded, as is the case in current commercially available FPGAs [63] [88]. An example of a two-level 16:1 multiplexer is shown in Figure 5.9(a). This is a more efficient implementation than a one-level implementation, as the multiplexers of the first level share control bits and hence less SRAM cells are required. For the implementation of a multiplexer in one level we assume the use of pass transistors and one hot encoding [88]. An example is shown in Figure 5.9(b) for a 4:1 multiplexer.

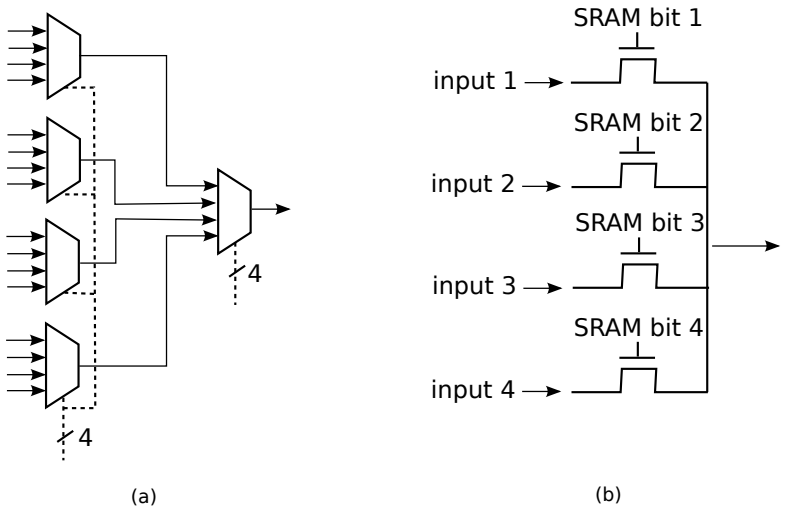


Figure 5.9: (a) An example of a two-level 16:1 multiplexer controlled by 8 SRAM bits. (b) Implementation of a 4:1 multiplexer using pass-transistors and one hot encoding.

The number of routing configuration bits that needs to be reconfigured using conventional DPR is given by

$$B_{conv}^{R} = \sum_{f \in F} b_f \tag{5.12}$$

where F is the set of routing configuration frames of the RR that contain at least one dynamic bit, b_f is the number of bits per routing frame given by Equation 5.13.

$$b_f = \sum_{m \in M_F} C_m \tag{5.13}$$

where M_F is the set of multiplexers that is controlled by the bits in frame F. I stress that F only includes routing frames that contain at least one dynamic bit. If a frame contains only static bits it is not counted in the configuration size.

For the StaticRoute tool flow the configuration can also be divided into a part for the CLBs and a part for the routing:

$$B_{new} = B_{new}^{CLB} + B_{new}^{R} \tag{5.14}$$

$$B_{new}^{CLB} = B_{total}^{CLB} \tag{5.15}$$

As mentioned above, in both the case of conventional DPR and the StaticRoute flow all CLBs in the reconfigurable region are rewritten. That is why B_{new}^{CLB} also equals B_{total}^{CLB} and is thus also given by Equation 5.10. However, in contrast with conventional DPR, the routing frames of the RR are divided into a set of static frames F_S and a set of dynamic frames F_D, with $F_S \bigcup F_D = F_{total}$. A reduction of reconfiguration overhead is achieved for the StaticRoute flow because only a much smaller dynamic part of the configuration memory needs reconfiguration. The content of the larger number of static frames is the same for all implemented circuits and therefore never needs to be rewritten in the configuration memory during run-time. Again, only routing frames that contain at least one dynamic bit are included in F_D. The size of the configuration of the interconnection network in the reconfigurable region for StaticRoute hence is

$$B_{new}^{R} = \sum_{f \in F_D} b_f \tag{5.16}$$

where b_f is the number of bits per routing frame and is calculated the same as above (i.e., using Equation 5.13). Finally, the reduction of reconfiguration overhead (RRO) is calculated as

$$RRO = \frac{(B_{conv} - B_{new})}{B_{conv}} = 1 - \frac{B_{new}}{B_{conv}}. \tag{5.17}$$

In Table 5.3 the number of configuration bits in the experiments with 2 circuits (case with 50% static SBs and 50% static CBs) is presented for B_{total}^{CLB}, B_{total}^{R}, B_{conv}^{R} and B_{new}^{R}. A column containing the fraction $\frac{B_{conv}^{R}}{B_{total}^{R}}$ was added. B_{total}^{R} is the total number of configuration bits present in the configuration memory of the routing of the reconfigurable region, independent of the fact whether the bits are static or dynamic. This column clearly shows that, in the case of conventional DPR, the dynamic bits are scattered over, on average, 94% of the frames of the reconfigurable region. Only 6% of the frames accidentally happen to have the same bit values for the different circuits. These are mostly frames that are not used by both circuits.

Table 5.3: The reduction in reconfiguration overhead (RRO) of the StaticRoute flow compared to the conventional DPR flow (in %), for the experiments with 2 circuits (case with 50% static SBs and 50% static CBs).

Exp	B_{total}^{CLB} (bits)	B_{total}^{R} (bits)	B_{conv}^{R} (bits)	$\frac{B_{conv}^{R}}{B_{total}^{R}}$ (%)	B_{new}^{R} (bits)	RRO (%)
0	135520	308800	289404	94	146198	34
1	542080	1229004	1192694	97	596126	34
2	542080	1229004	1183194	96	591250	34
3	362880	834968	813972	97	407024	35
4	219520	505660	462724	92	231890	34
5	323680	745342	729990	98	366768	34
6	592480	1339474	1253246	94	630004	34
7	161280	377924	354840	94	177130	34
8	592480	1339474	1255156	94	630736	34
9	448000	1023724	975330	95	484592	34
10	592480	1339474	1294606	97	649386	34
11	219520	505660	463256	92	233322	34
12	189280	439182	392408	89	196582	34
13	362880	834968	790432	95	392156	35
14	542080	1229004	1186302	97	593142	34
15	493920	1123754	1043826	93	524450	34
16	286720	663204	567530	86	288734	33
17	189280	439182	389976	89	197074	33
18	362880	834968	797634	96	397902	34
19	493920	1123754	1042322	93	523820	34
			Avg:	94	Avg:	34

The Altera IV configuration port that has a width of one byte and runs at 125 MHz [7]. Therefore by assuming a configuration bandwidth of 1 Gigabits per second, we can estimate the reconfiguration time in case of conventional DPR to be on average around 1.5 ms.

Table 5.3 corresponds to the case where 50% of the connection blocks and 50% of the switch blocks are marked static. The dynamic routing bits are clustered in the remaining dynamic part of the reconfigurable region. This results in a reduction of the routing reconfiguration overhead of around 50%. In Table 5.3 we see that this corresponds to a reduction of the total reconfiguration overhead (RRO) of 34% (due to the overhead for the CLBs being the same in both cases).

Columns RRO in Table 5.5 present the reduction of reconfiguration overhead for all the 16 cases (combinations of percentages of SB and CB marked static) considered in the experiments. These are values which are an average of the values obtained in the 20 experiments conducted for each case.

In the formulas above we see that, as the size of the static part increases, the size of the dynamic part decreases and this results in a larger reduction of reconfiguration overhead. Note that all CLBs will be rewritten completely during run-time. The actual reduction is therefore mostly dependent on the relative size of the configuration memory dedicated to the different components of the reconfigurable fabric, namely the CLBs, the connection blocks and the switch blocks.

For the FPGA architecture used in the experiments, 30% of the total number of configuration bits is used for CLBs, 38% for switch blocks and 32% for connection blocks. If, for example, none of the switch blocks and all connection blocks are marked static, this will result in about 32% reduction of the reconfiguration overhead. Marking 0% of the switch blocks and only 50% of the connection blocks static, results in a reduction of around 16%. The actual reduction is 14%. This can be clearly seen in Table 5.5.

Impact on maximum clock frequency In the StaticRoute flow the different circuits are not implemented separately, as is the case in the conventional DPR flow. Instead, the circuits are routed simultaneously using StaticRoute. In this section we assess the impact this has on the maximum clock frequency of the circuits. For each circuit we compare the implementation with the conventional DPR flow to the one resulting from using StaticRoute.

Table 5.4: The reduction in maximum clock frequency (Δ_i) for the experiments when using StaticRoute with 2 circuits, in % relative to conventional DPR (case with 50% static SBs and 50% static CBs).

Exp.	F_0	F_1	F_0^S	F_1^S	Δ_0	Δ_1
	(MHz)		(MHz)		(%)	
0	168	254	156	231	7	9
1	243	169	224	158	8	7
2	219	243	211	231	4	5
3	119	320	116	285	3	11
4	209	288	195	271	7	6
5	140	258	136	230	3	11
6	191	215	181	195	5	9
7	220	259	210	241	5	7
8	139	215	131	196	6	9
9	230	167	203	155	12	7
10	139	278	134	254	4	9
11	288	237	272	220	6	7
12	230	222	211	198	8	11
13	166	219	154	211	7	4
14	209	243	203	233	3	4
15	151	198	145	174	4	12
16	314	185	284	178	10	4
17	151	225	147	211	3	6
18	320	166	292	150	9	10
19	161	222	156	213	3	4
	Avg: 215		Avg: 200		Avg: 6	

Table 5.5: Average reduction of reconfiguration overhead (RRO) and average reduction in maximum clock frequency (RCF) for the StaticRoute flow, in % relative to conventional DPR.

		% static CB							
		0%		25%		50%		75%	
		RRO	RCF	RRO	RCF	RRO	RCF	RRO	RCF
% static SB	0%	0	-1	7	2	14	3	22	8
	25%	10	1	18	3	24	5	32	9
	50%	20	4	27	5	34	6	41	15
	75%	29	9	37	11	44	18	51	21

(in %)

The reduction of maximum clock frequency (RCF) for the 16 cases considered in the experiments are shown in column RCF of Table 5.5. The results shown here are each an average of the 20 experiments that have been carried out. The results depend on how the connection and switch blocks are marked static.

The actual clock frequencies of the experiments for the case of 50% static SBs and 50% static CBs are presented in Table 5.4. The F_i columns show the maximum clock frequencies for the implementations that use the conventional DPR tool flow. The maximum clock frequencies obtained in the StaticRoute tool flow, are denoted F_i^S. The reduction of maximum clock frequency, compared to the conventional DPR flow (in %), is shown in the columns Δ_i.

Figure 5.10 shows the values of Table 5.5 in a scatter plot. The pareto-optimal points are marked in black. Again, we see that the relative reduction in maximum clock frequency depends on the relative size of the static part. As the size of the static part increases, the reconfiguration overhead decreases and the maximum clock frequency decreases more. Up until a reduction in reconfiguration overhead of around 34% the impact on maximum clock frequency is less significant. When we cross this value, the maximum clock frequency decreases faster.

We can also see in Table 5.5 that it is better to spread the static part over the connection blocks and switch blocks. When, for example,

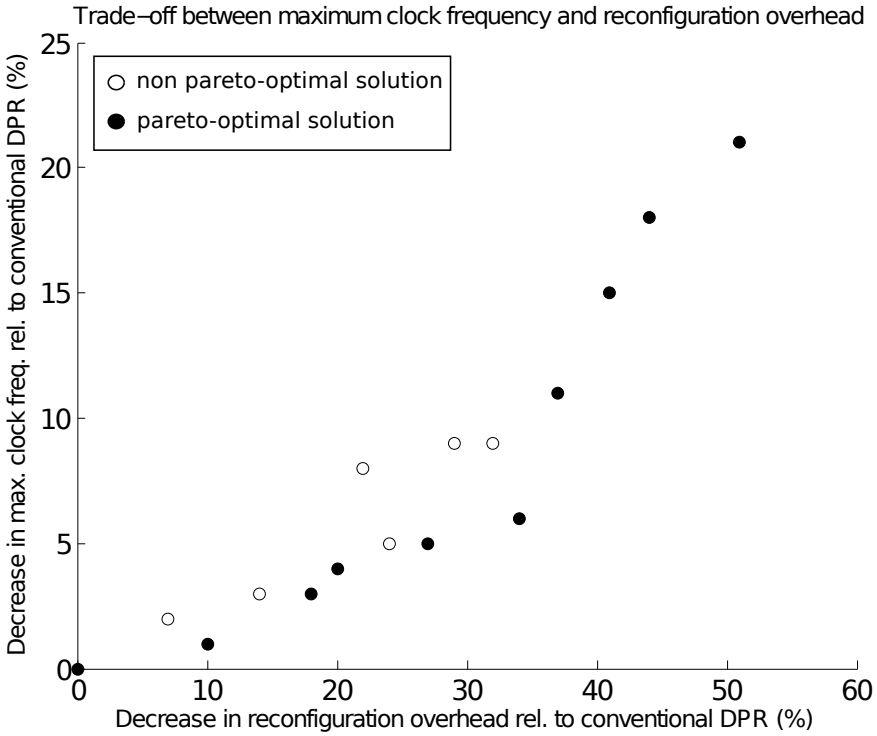


Figure 5.10: Trade-off between reconfiguration overhead and maximum clock frequency.

75% of the connection blocks are marked static then the reduction in maximum clock frequency is on average 8%. If, however, 50% of the connection blocks and 50% of the switch blocks are marked static, then the average reduction is smaller, namely 6%.

For this last case the reduction of reconfiguration overhead is even higher, as shown in Table 5.5. In Figure 5.10 and Table 5.5 we can also see that an implementation is more likely to be pareto-optimal if the static part is spread more evenly over the SBs and CBs.

We see that the discussion above is even more pronounced for the maximum values found in Table 5.6. The minimum values are also reported in Table 5.6. When the static part is small or zero we see that there are some small positive values. This means that in some cases StaticRoute turned out to have a slightly better maximum clock frequency than the PATHFINDER algorithm. This is possible because both algorithms are heuristics.

Table 5.6: Maximum and minimum reduction of maximum clock frequency (RCF), in % relative to conventional DPR.

		% static CB							
		0%		25%		50%		75%	
		Max	Min	Max	Min	Max	Min	Max	Min
% static SB	0%	2	-3	3	1	7	1	13	5
	25%	4	-2	5	1	9	4	18	6
	50%	9	0	10	2	12	3	24	8
	75%	14	4	19	6	31	9	37	8

(in %)

If the static part in the CBs or SBs is not chosen higher than 50%, then the reduction in maximum clock frequency is not higher than 6% on average and maximum 12%. The reduction is 10% on average and maximum 15%, if only the SBs or CBs have a static part of 75%. The reduction is 20% on average and maximum 37% if both CBs and SBs have a static part of 75%. As mentioned earlier, StaticRoute was not able to find a solution when all the CBs and/or SBs were completely marked static.

We can conclude that, for 2 circuits, the impact on the maximum clock frequency is limited if the static part in the SBs or CBs is not higher than 50%. It is also better to spread the static part evenly over the CBs and SBs. This way the same reduction of reconfiguration overhead is achieved (or more), while the impact on the maximum clock frequency is less significant.

It is difficult to say what the actual impact is of this reduction in maximum clock frequency on the application. This will depend on the type of application. There are applications that do not run at their maximum performance, because system requirements are not that stringent. Think, for example, of bandwidth-restricted applications. Also, since FPGAs are used a lot for parallel applications, they sometimes rely more on massive parallelism than on high clock frequencies for performance.

5.1.6 Implementing more than 2 circuits

The previous section dealt with how the static part is best selected in the configuration memory. Only implementations that time-multiplexed 2 circuits on the same reconfigurable region were considered. In this section we will look at what the overhead is when our new flow using StaticRoute implements more than 2 circuits into the same reconfigurable region.

5.1.6.1 Experimental set-up

The FPGA architecture used in this section is the same as the one described in Section 5.1.5.1.

The circuits used in the experiments for N = 2, N=3 and N= 4 are presented in Table 5.1 and Table 5.2 in Section 5.1.5.1. In the experiments StaticRoute was not able to find a DPR solution with static parts when N was greater than 4. The reason for this is probably that StaticRoute has a fixed number of routing iterations (which is the same as the conventional Pathfinder algorithm in VPR). StaticRoute doesn't seem to find a solution, given this number of iterations. However, for a more thorough exploration we refer to future work. The execution time of the algorithm is best optimized before the relation between the number of routing iterations and the number of circuits is researched more thoroughly.

Based on the results from the previous section, in this section only the case is considered where 50% of the SBs and 50% of the CBs were marked static.

5.1.6.2 Results

Impact on reconfiguration overhead The reduction of reconfiguration overhead is calculated as explained in Section 5.1.5.2. The results for the experiments with N = 2 were already presented in Table 5.3 in Section 5.1.5.2. The results for N = 3 and N = 4 are presented in Tables 5.7 and 5.8, respectively. Column RRO shows the reduction of reconfiguration overhead for each experiment. On the bottom of this column the average value is presented.

In these tables we see that the impact on reconfiguration overhead hardly depends on the number of implemented circuits, N.

Table 5.7: The reduction in reconfiguration overhead (RRO) for the experiments using StaticRoute with 3 circuits, in % relative to conventional DPR (case with 50% static SBs and 50% static CBs).

Exp	B_{total}^{CLB} (bits)	B_{total}^{R} (bits)	B_{conv}^{R} (bits)	$\frac{B_{conv}^{R}}{B_{total}^{R}}$ (%)	B_{new}^{R} (bits)	RRO (%)
0	362880	834968	814276	98	407936	35
1	542080	1229004	1183194	96	590566	34
2	362880	834968	784344	94	388518	35
3	448000	1023724	981674	96	491538	34
4	219520	505660	472676	93	237882	34
5	592480	1339474	1250760	93	627214	34
6	362880	834968	810054	97	406298	34
7	362880	834968	792518	95	393646	35
8	592480	1339474	1295450	97	649926	34
9	592480	1339474	1277194	95	641528	34
10	592480	1339474	1290064	96	644290	34
11	542080	1229004	1186834	97	595260	34
12	592480	1339474	1242988	93	625080	34
13	592480	1339474	1252764	94	631612	34
14	592480	1339474	1260918	94	635036	34
15	542080	1229004	1193506	97	598026	34
16	323680	745342	732460	98	367414	35
17	362880	834968	817792	98	410326	35
18	362880	834968	807504	97	405158	34
19	219520	505660	470966	93	235070	34
			Avg:	96	Avg:	34

Table 5.8: The reduction in reconfiguration overhead (RRO) for the experiments using StaticRoute with 4 circuits, in % relative to conventional DPR (case with 50% static SBs and 50% static CBs).

Exp	B_{total}^{CLB} (bits)	B_{total}^{R} (bits)	B_{conv}^{R} (bits)	$\frac{B_{conv}^{R}}{B_{total}^{R}}$ (%)	B_{new}^{R} (bits)	RRO (%)
0	323680	745342	730902	98	366002	35
1	592480	1339474	1267310	95	638668	34
2	448000	1023724	992204	97	496648	34
3	362880	834968	812182	97	406682	35
4	592480	1339474	1296204	97	651440	34
5	323680	745342	732460	98	367110	35
6	362880	834968	810054	97	406336	34
7	592480	1339474	1296026	97	652396	34
8	592480	1339474	1280810	96	649328	34
9	286720	663204	581308	88	293556	33
10	542080	1229004	1184598	96	591908	34
11	219520	505660	478984	95	239326	34
12	592480	1339474	1297576	97	653650	34
13	286720	663204	584550	88	295046	33
14	592480	1339474	1267918	95	638662	34
15	592480	1339474	1263064	94	634352	34
16	592480	1339474	1293554	97	653042	34
17	592480	1339474	1267744	95	637892	34
18	493920	1123754	1045530	93	529390	34
19	362880	834968	798992	96	399040	34
			Avg:	95	Avg:	34

Table 5.9: The reduction in maximum clock frequency (Δ_i) for the experiments when using StaticRoute with 3 circuits, in % relative to conventional DPR (case with 50% static SBs and 50% static CBs).

Exp.	F_0	F_1 (MHz)	F_2	F_0^S	F_1^S (MHz)	F_2^S	Δ_0	Δ_1 (%)	Δ_2
0	219	119	320	198	112	313	10	6	2
1	207	243	201	186	224	187	10	8	7
2	166	176	183	151	148	170	9	16	7
3	258	207	167	220	175	141	15	15	16
4	179	320	219	168	294	188	6	8	14
5	191	181	226	180	167	196	6	8	13
6	250	176	119	213	156	109	15	11	8
7	213	166	219	178	156	198	16	6	10
8	278	191	313	267	176	282	4	8	10
9	313	135	139	271	130	133	13	4	4
10	191	200	135	179	188	124	6	6	8
11	116	243	294	109	235	272	6	3	7
12	255	191	172	220	179	156	14	6	9
13	118	156	191	107	140	179	9	10	6
14	139	215	213	132	196	196	5	9	8
15	243	142	294	228	137	275	6	4	6
16	226	140	306	213	131	267	6	6	13
17	223	119	278	194	112	263	13	6	5
18	250	278	166	216	238	157	14	14	5
19	288	174	320	272	151	272	6	13	15
		Avg: 210			Avg: 190			Avg: 9	

Table 5.10: The reduction in maximum clock frequency (Δ_i) for the experiments when using StaticRoute with 4 circuits, in % relative to conventional DPR (case with 50% static SBs and 50% static CBs).

Exp.	F_0	F_1	F_2	F_3	F_0^S	F_1^S	F_2^S	F_3^S	Δ_0	Δ_1	Δ_2	Δ_3
	(MHz)				(MHz)				(%)			
0	140	190	195	226	130	154	150	204	7	19	23	10
1	213	139	226	320	174	115	202	288	18	17	11	10
2	190	167	313	148	166	153	243	131	13	8	22	11
3	335	213	119	223	269	166	107	191	20	22	10	14
4	278	118	139	135	251	101	125	118	10	14	10	13
5	195	215	174	140	179	191	156	130	8	11	10	7
6	176	119	223	250	139	104	200	217	21	13	10	13
7	191	215	278	172	177	178	250	138	7	17	10	20
8	191	200	255	320	156	166	220	247	18	17	14	23
9	210	314	237	225	192	254	214	201	9	19	10	11
10	294	193	243	201	245	167	222	181	17	13	9	10
11	288	237	320	226	267	195	258	201	7	18	19	11
12	156	200	191	278	127	169	150	226	19	16	21	19
13	210	226	300	314	186	191	251	245	11	15	16	22
14	255	320	213	139	219	272	176	122	14	15	17	12
15	139	226	181	172	118	176	152	141	15	22	16	18
16	226	191	255	278	192	160	220	250	15	16	14	10
17	139	215	161	255	115	183	138	216	17	15	14	15
18	204	241	151	226	173	210	126	198	15	13	17	12
19	200	176	149	166	161	147	130	145	20	16	13	13
		Avg: 214				Avg: 182				Avg: 15		

As we mentioned before, the reduction of reconfiguration overhead is mostly dependent on the relative size of the configuration memory dedicated to the different components of the reconfigurable fabric, namely the CLBs, the connection blocks and the switch blocks. We again added a column that shows the fraction $\frac{B^R_{conv}}{B^R_{total}}$. This column clearly shows that, in the case of conventional DPR, the dynamic bits are scattered over around 95% of the frames of the reconfigurable region.

In the case of StaticRoute the static part does not need to be rewritten during run-time. In the experiments in this section the fraction of static SBs and static CBs were both chosen to be 50%. This of course results in a reduction of *routing* reconfiguration overhead of around 50%. In Tables 5.3, 5.7 and 5.8 we see that this corresponds to a significant reduction in *total* reconfiguration overhead of 34%.

Impact on maximum clock frequency In this section we take a look at how the maximum clock frequency is affected as the number of circuits N increases. As mentioned earlier, StaticRoute was not able to find a DPR solution with static parts when N is greater than 4.

The results obtained in all the experiments for this section can be found in Table 5.4 for N = 2 and in Tables 5.9 and 5.10 for N = 3 and N = 4, respectively. In these tables the maximum clock frequencies obtained using the conventional DPR flow are denoted as F_i. The ones obtained using StaticRoute were denoted as F_i^S. The last columns of these tables show the difference (Δ_i) between the corresponding maximum clock frequencies, i.e. the reduction of maximum clock frequency (RCF).

Figure 5.11 shows the average values of these tables, namely the average reduction of maximum clock frequency compared to conventional DPR. Also the maxima and minima of the experiments are shown on this figure using error bars. As discussed in Section 5.1.5.2, the decrease is 6% on average for 2 circuits. The results show that the reduction is higher as the number of circuits N increases. It is 9% on average for 3 circuits and 15% for 4 circuits. We see that the maximum values follow the same trend. They are 12% for 2 circuits and increase to 16% for 3 circuits and 23% for 4 circuits. The minimum values indicate that there are also some circuits that are barely influenced by implementation using the StaticRoute flow.

In fact, the analysis made above assumes the clock frequency

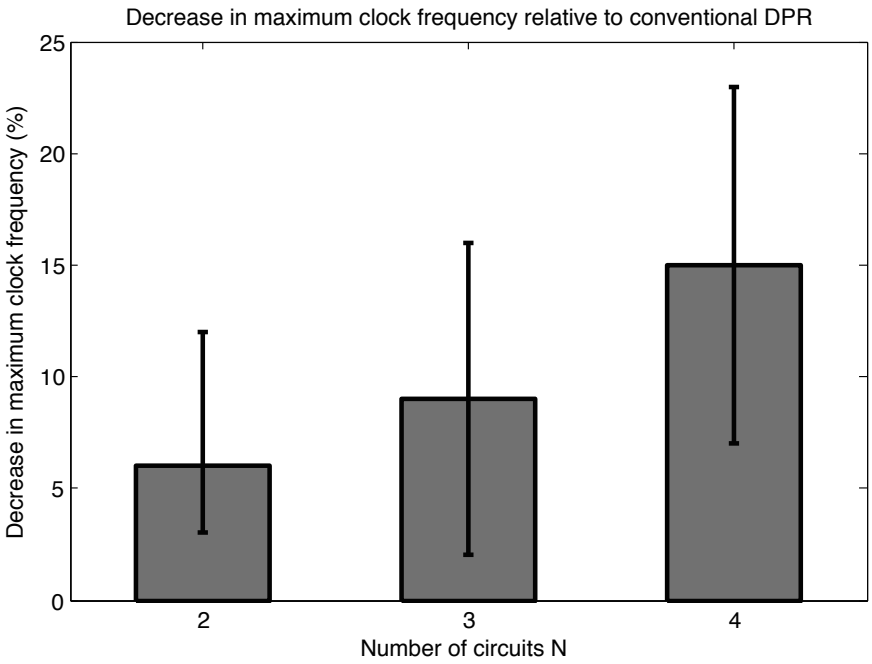


Figure 5.11: Averages of the reduction of maximum clock frequency of the StaticRoute flow (in % rel. to conventional DPR), as a function of the number of circuits N. Maxima and minima of the experiments are indicated with error bars.

changes when the circuit implemented in the reconfigurable region changes. Although this is theoretically possible, it is not done in practice as it significantly increases the complexity of the design. It therefore is reasonable to assume that the clock frequency remains fixed when switching between circuits [20]. In Table 5.11 the new values of the reduction in clock frequency are presented. The values are very similar as the ones presented before, but improve with a few percent. The RCF is 5% on average for 2 circuits. The reduction is higher as the number of circuits N increases. It is 8% on average for 3 circuits and 14% for 4 circuits.

Impact on off-line compilation time of the tool flow As we can see in Figure 5.4 the StaticRoute tool flow reuses the first steps of the conventional flow. The main difference compared to the con-

Table 5.11: Overview of the reduction of the clock frequency (RCF) (in %) relative to conventional DPR for the StaticRoute (SR) and ClusterRoute (CR) algorithm, for increasing number of circuits.

	Avg.		Max.		Min.	
Nr. of circuits	SR	CR	SR	CR	SR	CR
2	-5	-5	-11	-10	-3	-2
3	-8	-11	-16	-19	-4	-3
4	-14	-21	-20	-31	-7	-12

(in %)

ventional DPR flow is the routing step. In the StaticRoute flow the circuits are routed simultaneously to also be able to take the reconfiguration overhead into account. Calculating this cost results in a higher compilation time of the tool flow.

Table 5.12 presents the compilation times of the experiments of the conventional DPR flow. The timing experiments are all done using an Intel Core i7-3770 Quad Core processor running at 3.40 GHz with 16 GiB of memory running the Java HotSpotTM 64-Bit Server VM. Because the circuits are available in blif format, these compilation times do not include the time needed for synthesis.

In the conventional DPR flow the different circuits are implemented separately and there are a limited number of circuits. To allow a fair comparison we assume that this flow can easily be run on a multi-core processor. That is why we first measure the time needed to implement each circuit and then take the maximum value.

Table 5.13 presents the slowdown of the experiments when using this flow compared to the conventional DPR flow. In these measurements we assumed the first steps of the flow are done in parallel, just as in the conventional flow. The slowdown of the routing step is 11× for 2 circuits and increases to 16× for 4 circuits. The slowdown of the complete flow is smaller, 7× for 2 circuits and increases to 10× for 4 circuits.

There are 3 main reasons for this slowdown. First, in the StaticRoute flow the compilation time increases because a cost needs to be

Table 5.12: Compilation time of the experiments in the conventional DPR flow (in seconds).

	N = 2		N = 3		N = 4	
Experiment	Routing	Total	Routing	Total	Routing	Total
0	3.52	7.79	34.15	54.61	16.18	34.71
1	100.67	127.29	32.54	62.97	92.37	132.46
2	27.69	55.91	37.15	57.02	88.22	113.86
3	34.34	54.75	105.42	131.75	34.14	54.72
4	3.43	11.22	7.63	14.89	92.19	132.26
5	16.19	34.73	33.49	65.54	15.94	34.55
6	40.65	75.01	33.9	54.49	27.93	47.58
7	4.67	9.04	42.22	62.55	33.49	65.58
8	78.27	117.26	34.11	66.84	39.71	74.02
9	106.12	132.26	92.38	132.43	8.36	14.24
10	77.73	117.02	33.34	65.42	27.91	56.44
11	3.43	11.16	46.62	67.53	3.41	11.54
12	6.66	13.09	34.29	66.62	90.04	119.86
13	42.21	62.38	92.11	122.17	9.77	15.65
14	31.22	61.39	94.15	134.32	87.43	127.6
15	72.16	100.5	73.63	102.95	76.7	115.51
16	8.02	15.69	16.32	34.92	40.52	74.95
17	6.64	13.06	34.39	54.97	111.8	138.47
18	41.88	62.05	37.61	57.56	72.73	101.24
19	71.92	100.72	3.44	11.57	42.31	62.49

(in seconds)

Table 5.13: Slowdown of the compilation when using the StaticRoute flow, compared to the conventional DPR flow.

	N = 2		N = 3		N = 4	
Experiment	Routing	Total	Routing	Total	Routing	Total
0	14	7	11	7	16	8
1	9	7	13	7	9	6
2	11	6	11	8	12	9
3	10	6	8	7	14	9
4	13	5	18	9	14	10
5	12	6	16	9	12	6
6	8	5	10	7	15	9
7	16	9	10	7	18	10
8	8	6	15	8	20	11
9	7	6	10	7	25	15
10	10	7	24	13	20	10
11	17	6	15	11	32	10
12	14	8	13	7	15	12
13	8	6	14	11	27	17
14	9	5	8	6	10	7
15	9	7	13	10	10	7
16	17	9	13	6	16	9
17	13	7	13	9	13	11
18	9	6	12	8	10	7
19	8	6	23	8	12	8
Average:	11×	7×	13×	8×	16×	10×

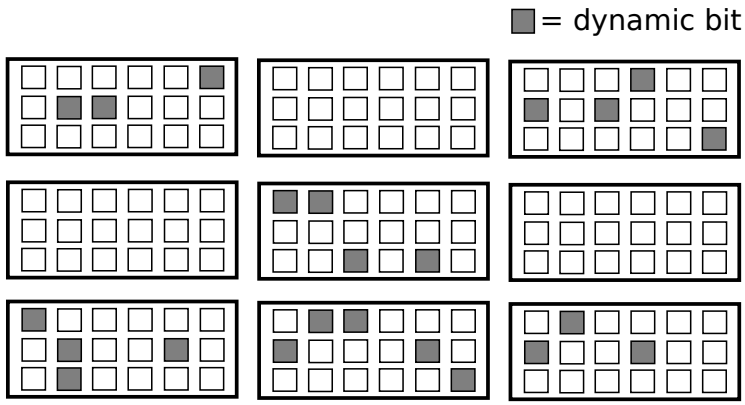


Figure 5.12: Clustering of the dynamic bits in fewer configuration frames when using the ClusterRoute tool flow.

calculated that takes the reconfiguration overhead into account. This cost is calculated in the inner loop of the PATHFINDER algorithm. Second, the routing step does no longer happen in parallel. Finally, our implementation of StaticRoute is not thoroughly optimized, as is the case for the router in VTR.

We note that both the conventional flow and the StaticRoute flow are run off-line. The compilation time of the flow therefore does not affect the performance of the application on the FPGA.

StaticRoute explores the RRG in the same way as the PATHFINDER algorithm. Extra time is needed per explored wire node to calculate the wire congestion. The scaling behaviour of the StaticRoute algorithm is therefore the same as that of PATHFINDER [71].

5.2 ClusterRoute: Automatic clustering of dynamic bits

In the previous section a new router for low-overhead limited-context DPR, called StaticRoute, has been presented. In the StaticRoute tool flow, shown in Figure 5.4 an extra step is needed before routing, where a portion of the routing network is marked as being static. The result of this step is that the configuration frames are split up in frames marked static and frames marked dynamic. StaticRoute

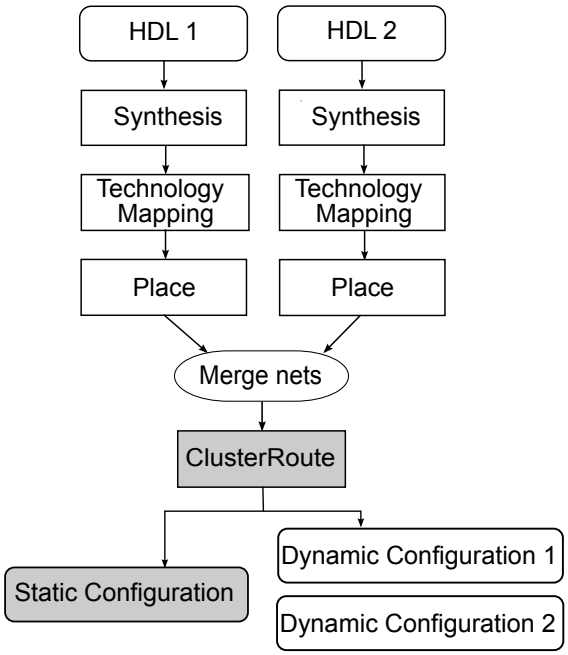


Figure 5.13: The ClusterRoute tool flow.

routes the nets of the circuits in such a way that the dynamic bits are clustered in the dynamic frames, as shown in Figure 5.3 .

In this section a router, called ClusterRoute, is presented where marking routing switches as static is no longer necessary. ClusterRoute automatically routes the interconnections in such a way that the dynamic bits are clustered in fewer configuration frames, as is shown in Figure 5.12. ClusterRoute is also based on the PATHFINDER algorithm [13] [71], but takes into account both the wire length and the reconfiguration overhead.

5.2.1 ClusterRoute tool flow

The tool flow that uses CluserRoute is shown in Figure 5.13. As can be seen in the figure, the flow is very similar to the StaticRoute flow, depicted in Figure 5.4. However, marking routing switches as static is no longer necessary. The flow implements the circuits in the reconfigurable region completely automatic. As the StaticRoute flow, it generates a static configuration that only needs to be loaded in the

configuration memory at start-up. It also generates a dynamic configuration for each of the circuits. These can be used to configure the dynamic configuration frames of the FPGA during start-up and runtime. Because the dynamic configurations are much smaller than a complete configuration of the reconfigurable region, the reconfiguration overhead is considerably reduced.

ClusterRoute is also based on the PATHFINDER algorithm, uses an extended routing resource graph during routing and detects dynamic bits in the same way as StaticRoute, which is explained in Section 5.1.2 and Section 5.1.3. ClusterRoute, however, has another cost function that makes it possible to automatically implement circuits in the reconfigurable region, such that less dynamic routing configuration frames are generated. This cost function is described in the following section.

5.2.2 Cost function

In this section I describe how I extended the cost function used in the PATHFINDER algorithm to also take the reconfiguration overhead into consideration. As was the case for StaticRoute, to take the reconfiguration overhead into consideration an extra cost per wire is added, called the reconfiguration cost in the case of ClusterRoute. In our novel cost function, the cost of a node in the extended RRG is thus

$$cost(n,c) = \begin{cases} cost_w(n,c) + cost_r(n) & \text{if } n \text{ is a wire} \\ 0 & \text{if } n \text{ is a switch} \end{cases} \tag{5.18}$$

where $cost_w(n,c)$ is the wire congestion cost associated with wire node n and circuit c. $Cost_r(n)$ is the reconfiguration cost associated with wire node n.

The term $cost_w(n,c)$ is given by Equation 5.4, already presented in Section 5.1.4 for StaticRoute.

The term $cost_r(n)$ takes the reconfiguration overhead into consideration. The reconfiguration cost is calculated based on the union of the fan-in switch nodes of the current wire node n and the fan-out switch nodes of the previous wire node in the routing path currently being evaluated. As was explained in Section 5.1.4, this set of switch nodes is called $S(n)$. Figure 5.7 shows an example where

the switches of $S(n)$, associated with a wire node n, are identified in black.

Given a wire node n, with its associated set of switches $S(n)$, we propose the following equation for the reconfiguration cost

$$cost_r(n) = \sum_{s \in S(n)} cost_r(s), \tag{5.19}$$

where $cost_r(s)$ is the reconfiguration cost of one switch s and is given by:

$$cost_r(s) = \begin{cases} 0 & \text{if } N_d(F(s)) = 0 \\ (1 - \frac{N_d(F(s))}{T(F(s))}) \cdot factor & \text{if } 0 < N_d(F(s)) \leq threshold \\ 1 - \frac{N_d(F(s))}{T(F(s))} & \text{otherwise} \end{cases} \tag{5.20}$$

where $F(s)$ is the configuration frame to which the switch s belongs. This cost function is also depicted in Figure 5.14. $N_d(F(s))$ and $T(F(s))$ are the number of dynamic bits and the total number of bits in this frame, respectively.

The reconfiguration cost is basically inversely proportional to the number of dynamic bits in the frame. When the number of dynamic bits in the frame increases, each bit is associated with a lower cost. This way the dynamic bits are clustered in less frames. The parameters $factor$ and $threshold$ were introduced to associate an extra cost with frames with only a few dynamic bits. This is done by multiplying the base cost, which is a basic inverse linear function, with a factor when the number of dynamic bits in the frame is lower than a certain threshold. This stimulates the router to make these frames completely static. It also increases the cost to make a static frame dynamic. Based on a parameter sweep with 2 circuits, $factor$ and $threshold$ were both set to 4. For these parameter values a reduction of reconfiguration overhead of 31% was achieved, while the reduction of maximum clock frequency was on average limited to 5%.

5.2.3 Experimental set-up

The experimental set-up is mostly the same as for StaticRoute, as described in Section 5.1.5.1. ClusterRoute is also integrated in the VTR framework and based on the wire-length driven router [80]. The configuration frames are also chosen the same. Only in the case of Clus-

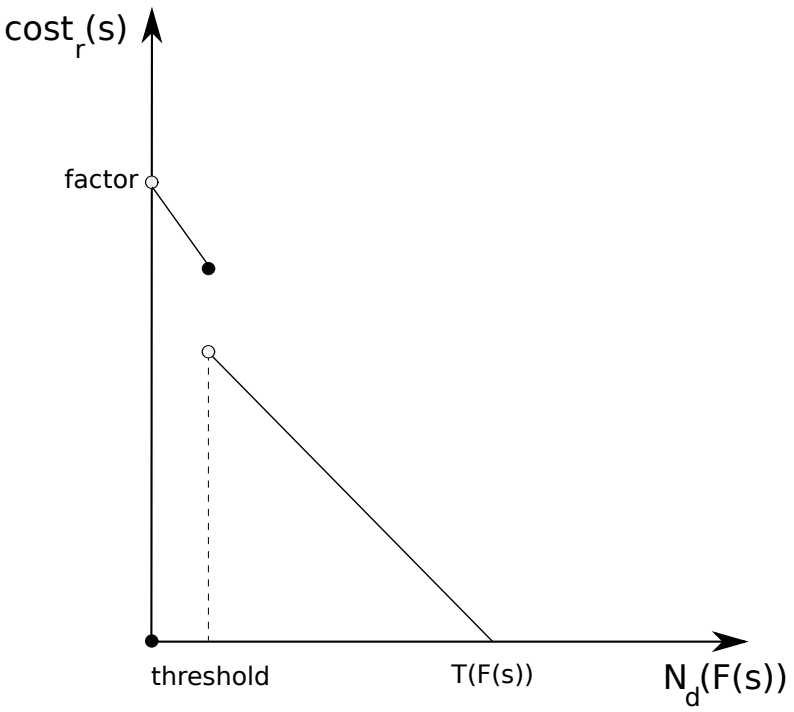


Figure 5.14: The reconfiguration cost associated with one switch s.

terRoute it is not necessary to mark switches as static. The circuits used in the experiments are also the same and can also be found in Table 5.1 and Table 5.2. ClusterRoute was also not able to find a solution when N was greater than 4. A discussion of the reason for this, can be found in Section 5.1.6.1. The reconfiguration overhead is also calculated as described in Section 5.1.5.2. The maximum clock frequency of the implemented circuits was also extracted using the timing analyser in the VTR framework.

5.2.4 Results

5.2.4.1 Impact on reconfiguration overhead

In Tables 5.14, 5.15 and 5.16 the results regarding the reconfiguration overhead can be found for the experiments where the number of circuits N was 2, 3 and 4, respectively.

In these tables the values obtained in the experiments are presented for B^{CLB}_{total}, B^{R}_{total}, B^{R}_{conv}, B^{R}_{new}. In Table 5.14 we see that an average reduction of reconfiguration overhead (RRO) is achieved of 32% for N = 2. This reduction of reconfiguration overhead decreases to 27% and 19% for N = 3 and N = 4, respectively.

5.2.4.2 Impact on maximum clock frequency

For each circuit we compare the implementation with the conventional DPR flow to the one resulting from using ClusterRoute. The clock frequencies of the experiments for the case of N = 2, N =3 and N = 4 are presented in Table 5.17, Table 5.18 and Table 5.19, respectively. As mentioned earlier, ClusterRoute was not able to find a solution when N is greater than 4. The F_i columns show the clock frequencies for the implementations that use the conventional DPR tool flow. Clock frequencies obtained in the ClusterRoute flow, are denoted with F^C_i. The last columns of these tables show the reduction of maximum clock frequency.

It can be seen that the decrease is 5% on average for 2 circuits. The results show that the decrease is higher as the number of circuits N increases. It is 13% on average for 3 circuits and 22% for 4 circuits.

Finally, Figure 5.15 shows these average values together with the maxima and minima of the experiments, using error bars. We see that the maximum values follow the same trend. It is 14% for 2 circuits and increases to 21% for 3 circuits and 31% for 4 circuits.

Table 5.14: The reduction in reconfiguration overhead (RRO) of the ClusterRoute flow compared to the conventional DPR flow (in %), for the experiments with 2 circuits.

Exp	B^{CLB}_{total} (bits)	B^{R}_{total} (bits)	B^{R}_{conv} (bits)	$\frac{B^{R}_{conv}}{B^{R}_{total}}$ (%)	B^{R}_{new} (bits)	RRO (%)
0	135520	308800	289404	94	180012	26
1	542080	1229004	1192694	97	851936	20
2	542080	1229004	1183194	96	445372	43
3	362880	834968	813972	97	415096	34
4	219520	505660	462724	92	265410	29
5	323680	745342	729990	98	359198	35
6	592480	1339474	1253246	94	617582	34
7	161280	377924	354840	94	220596	26
8	592480	1339474	1255156	94	616426	35
9	448000	1023724	975330	95	495472	34
10	592480	1339474	1294606	97	588670	37
11	219520	505660	463256	92	244460	32
12	189280	439182	392408	89	299918	16
13	362880	834968	790432	95	372012	36
14	542080	1229004	1186302	97	487464	40
15	493920	1123754	1043826	93	496078	36
16	286720	663204	567530	86	248170	37
17	189280	439182	389976	89	234000	27
18	362880	834968	797634	96	435656	31
19	493920	1123754	1042322	93	619798	28
			Avg:	94	Avg:	32

Table 5.15: The reduction in reconfiguration overhead (RRO) of the ClusterRoute flow compared to the conventional DPR flow (in %), for the experiments with 3 circuits.

Exp	B^{CLB}_{total} (bits)	B^{R}_{total} (bits)	B^{R}_{conv} (bits)	$\frac{B^{R}_{conv}}{B^{R}_{total}}$ (%)	B^{R}_{new} (bits)	RRO (%)
0	362880	834968	814276	98	427312	33
1	542080	1229004	1183194	96	691082	29
2	362880	834968	784344	94	468462	28
3	448000	1023724	981674	96	725312	18
4	219520	505660	472676	93	400592	10
5	592480	1339474	1250760	93	653650	32
6	362880	834968	810054	97	500496	26
7	362880	834968	792518	95	540354	22
8	592480	1339474	1295450	97	653236	34
9	592480	1339474	1277194	95	757782	28
10	592480	1339474	1290064	96	723444	30
11	542080	1229004	1186834	97	610594	33
12	592480	1339474	1242988	93	627144	34
13	592480	1339474	1252764	94	901706	19
14	592480	1339474	1260918	94	753970	27
15	542080	1229004	1193506	97	879072	18
16	323680	745342	732460	98	481618	24
17	362880	834968	817792	98	475556	29
18	362880	834968	807504	97	492256	27
19	219520	505660	470966	93	268914	29
			Avg:	96	Avg:	27

Table 5.16: The reduction in reconfiguration overhead (RRO) of the ClusterRoute flow compared to the conventional DPR flow (in %), for the experiments with 4 circuits.

Exp	B^{CLB}_{total} (bits)	B^{R}_{total} (bits)	B^{R}_{conv} (bits)	$\frac{B^{R}_{conv}}{B^{R}_{total}}$ (%)	B^{R}_{new} (bits)	RRO (%)
0	323680	745342	730902	98	527514	19
1	592480	1339474	1267310	95	748766	28
2	448000	1023724	992204	97	767844	16
3	362880	834968	812182	97	554662	22
4	592480	1339474	1296204	97	1053520	13
5	323680	745342	732460	98	555434	17
6	362880	834968	810054	97	507740	26
7	592480	1339474	1296026	97	785346	27
8	592480	1339474	1280810	96	1057512	12
9	286720	663204	581308	88	439264	16
10	542080	1229004	1184598	96	771536	24
11	219520	505660	478984	95	372476	15
12	592480	1339474	1297576	97	905578	21
13	286720	663204	584550	88	477840	12
14	592480	1339474	1267918	95	799100	25
15	592480	1339474	1263064	94	847902	22
16	592480	1339474	1293554	97	837618	24
17	592480	1339474	1267744	95	1103654	9
18	493920	1123754	1045530	93	686714	23
19	362880	834968	798992	96	720234	7
			Avg:	95	Avg:	19

Table 5.17: The reduction in maximum clock frequency (Δ_i) for the experiments when using ClusterRoute with 2 circuits, in % relative to conventional DPR.

Exp.	F_0	F_1	F_0^C	F_1^C	Δ_0	Δ_1
	(MHz)		(MHz)		(%)	
0	168	254	156	236	8	7
1	243	169	237	162	3	4
2	219	243	209	237	5	3
3	119	320	117	312	2	2
4	209	288	200	280	4	3
5	140	258	137	251	2	3
6	191	215	171	204	10	5
7	220	259	205	236	7	9
8	139	215	131	209	5	3
9	230	167	225	157	2	6
10	139	278	132	247	5	11
11	288	237	248	222	14	6
12	230	222	221	218	4	2
13	166	219	154	215	7	2
14	209	243	202	229	3	6
15	151	198	143	193	5	2
16	314	185	308	180	2	3
17	151	225	138	215	9	5
18	320	166	283	149	11	10
19	161	222	158	216	2	3
	Avg: 215		Avg: 204		Avg: 5	

Table 5.18: The reduction in maximum clock frequency (Δ_i) for the experiments when using ClusterRoute with 3 circuits, in % relative to conventional DPR.

Exp.	F_0	F_1 (MHz)	F_2	F_0^C	F_1^C (MHz)	F_2^C	Δ_0	Δ_1 (%)	Δ_2
0	219	119	320	191	103	283	13	13	12
1	207	243	201	184	200	176	11	18	13
2	166	176	183	150	150	164	10	15	11
3	258	207	167	243	178	141	6	14	15
4	179	320	219	168	260	190	6	19	13
5	191	181	226	181	161	214	5	11	5
6	250	176	119	212	157	109	15	11	8
7	213	166	219	188	140	199	12	16	9
8	278	191	313	241	167	248	13	13	21
9	313	135	139	253	116	118	19	14	15
10	191	200	135	163	174	122	15	13	10
11	116	243	294	104	213	255	10	12	13
12	255	191	172	227	169	141	11	11	18
13	118	156	191	111	135	171	6	14	11
14	139	215	213	121	193	188	13	10	12
15	243	142	294	213	116	243	12	19	17
16	226	140	306	191	132	265	16	6	13
17	223	119	278	192	115	234	14	3	16
18	250	278	166	219	226	148	12	19	11
19	288	174	320	238	155	262	18	11	18
		Avg: 210			Avg: 182			Avg: 13	

Table 5.19: The reduction in maximum clock frequency (Δ_i) for the experiments when using ClusterRoute with 4 circuits, in % relative to conventional DPR.

Exp.	F_0	F_1	F_2	F_3	F_0^C	F_1^C	F_2^C	F_3^S	Δ_0	Δ_1	Δ_2	Δ_3
	(MHz)				(MHz)				(%)			
0	140	190	195	226	115	149	152	183	18	22	22	19
1	213	139	226	320	160	117	189	247	25	16	17	23
2	190	167	313	148	153	117	216	126	19	30	31	15
3	335	213	119	223	290	164	101	168	13	23	15	25
4	278	118	139	135	226	89	112	102	19	25	19	24
5	195	215	174	140	153	174	124	123	21	19	28	12
6	176	119	223	250	139	99	183	189	21	17	18	24
7	191	215	278	172	152	158	229	140	20	27	18	19
8	191	200	255	320	148	169	203	253	22	15	21	21
9	210	314	237	225	163	243	183	175	22	23	23	22
10	294	193	243	201	212	143	201	150	28	26	17	25
11	288	237	320	226	199	181	224	175	31	24	30	23
12	156	200	191	278	107	142	146	208	31	29	24	25
13	210	226	300	314	153	171	227	217	27	24	24	31
14	255	320	213	139	203	263	156	114	20	18	27	18
15	139	226	181	172	113	195	154	135	19	14	15	21
16	226	191	255	278	180	164	203	229	20	14	21	18
17	139	215	161	255	112	179	118	193	20	17	27	24
18	204	241	151	226	170	197	116	183	17	18	23	19
19	200	176	149	166	168	139	113	114	16	21	24	31
		Avg: 214				Avg: 167				Avg: 22		

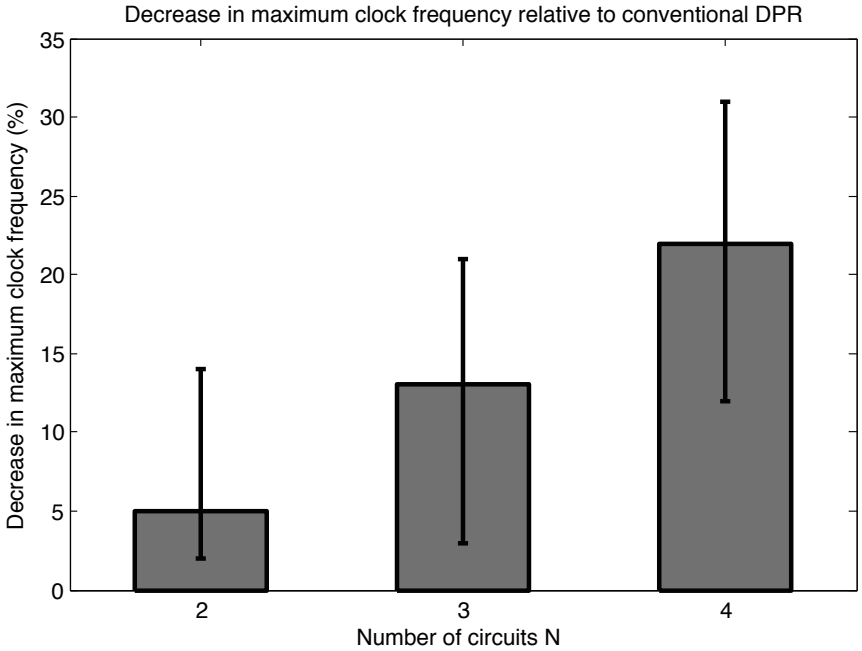


Figure 5.15: Averages of the reduction of maximum clock frequency of the ClusterRoute flow (in % rel. to conventional DPR), as a function of the number of circuits N. Maxima and minima of the experiments are indicated with error bars.

The minimum values indicate that there are also some circuits that are barely influenced by implementation using the ClusterRoute flow.

As discussed earlier, it is reasonable to assume that the clock frequency remains fixed when switching between circuits [20]. In Table 5.11 the new values of the reduction in clock frequency with this assumption are presented. The results for the ClusterRoute algorithm are indicated with 'CR'. The RCF for the ClusterRoute algorithm is 5% on average for 2 circuits. The reduction is again higher as the number of circuits N increases. It is 11% on average for 3 circuits and 21% for 4 circuits.

Table 5.20: Slowdown of the execution when using the ClusterRoute flow, compared to the conventional DPR flow.

	N = 2		N = 3		N = 4	
Experiment	Routing	Total	Routing	Total	Routing	Total
0	38	18	25	16	42	20
1	21	17	29	15	22	16
2	24	12	25	17	24	19
3	21	13	15	12	37	23
4	36	12	46	24	35	25
5	23	11	38	20	37	18
6	16	9	24	15	41	24
7	43	23	21	15	32	17
8	18	12	35	18	45	24
9	13	11	23	16	67	40
10	21	14	57	29	44	22
11	48	16	35	25	103	31
12	34	18	27	15	36	27
13	16	11	31	24	70	44
14	16	9	16	11	21	14
15	18	13	27	20	23	16
16	37	20	38	18	32	18
17	35	18	29	19	31	25
18	18	12	26	17	23	17
19	17	13	77	24	28	19
Average:	26×	14×	32×	18×	40×	23×

5.2.4.3 Impact on off-line compilation time of the tool flow

The timing experiments in this section were done using the same set-up as for the StaticRoute flow, as described in Section 5.1.6.2. Table 5.12 presents the compilation times of the experiments of the conventional DPR flow. Table 5.20 presents the slowdown of the experiments when using the ClusterRoute flow compared to the conventional DPR flow. The slowdown of the routing step is 26× for 2 circuits and increases to 40× for 4 circuits. The slowdown of the complete flow is smaller, 14× for 2 circuits and increases to 23× for 4 circuits

The reasons for this slowdown are the same as for the StaticRoute flow and were discussed in Section 5.1.6.2. Again, we note that both the conventional flow and our new flow are run off-line. The compilation time of the flow therefore does not affect the performance.

ClusterRoute explores the RRG in the same way as the PATHFINDER algorithm. Extra time is needed per explored wire node to calculate the reconfiguration cost. The scaling behaviour of the ClusterRoute algorithm is therefore the same as that of PATHFINDER [71].

5.3 Industrial relevance

As stated earlier, the use of Dynamic Partial Reconfiguration (DPR) of FPGAs can possibly result in an implementation that is more area efficient than a static approach. A higher area efficiency possibly results in the use of an FPGA with a lower cost. The downside of DPR is that a cost is associated with switching between configurations during run-time.

As the introduction of DPR impacts both the area and the execution time, capturing both the effects requires a metric for the area-time-efficiency. A measure that satisfies these conditions is the functional density: It is the number of computations (N) per unit of area (A) and per unit of time (T) [34]. It is also used to assess the benefits of dynamic reconfiguration implementations in [98].

$$FD = \frac{N}{T \cdot A} \tag{5.21}$$

The functional density depends heavily on the rate at which the application performs reconfigurations during run-time. In Figure 5.12 the functional density is plotted for the static, conventional

DPR, StaticRoute and ClusterRoute approach. To make this figure a configuration bandwidth of 1 Gigabits per second is assumed [7]. This corresponds with the Altera IV configuration port that has a width of one byte and runs at 125 MHz. The area, clock frequency and reconfiguration time are averaged over all 20 experiments considered in this chapter. The less optimistic clock frequency comparison is used, to show that even in this case the techniques in this chapter improve the functional density for a certain, more dynamic, configuration rate. A typical configuration time of a complete FPGA chip is around 10s of milliseconds [7]. The reconfiguration for partial reconfiguration is typically in the range of a few milliseconds. The average reconfiguration time in our experiments using an architecture based on the Stratix IV would be around 1.5 ms in the case of conventional DPR. When using StaticRoute the average reconfiguration time would drop to around 1 ms.

In the figure it can be seen that, when reconfiguration only happens once-in-a-while, it could be that the static implementation is not efficient, because there is hardware present that is not used for a significant portion of the execution time. In this case the approaches that use DPR will be more efficient since they require much less area and all of the hardware resources are used at all time. The conventional DPR approach is most efficient since the clock frequency at which the circuits operate is maximal and the area minimal, while the reconfiguration time plays less of a role as the reconfiguration rate is very low.

When the reconfiguration rate is very high, the cost of reconfiguration outweighs the benefits and a static implementation will be more efficient. When the configuration rate is in between (circuit reuse of around 500 thousand clock cycles) the StaticRoute approach will be more efficient. The StaticRoute algorithm always outperforms the ClusterRoute algorithm what functional density is concerned. Note however, that the ClusterRoute algorithm is more automated. If an application is constrained in area and the use of DPR is necessary, the StaticRoute approach clearly outperforms conventional DPR.

As explained above, the techniques in this chapter target applications that are more dynamic and thus where the reconfiguration overhead plays an important role. This can for example be an application that process network traffic. An example of such an application is shown in Figure 5.16 [76]. In this application the network packets are divided in complementary sets, based on the network

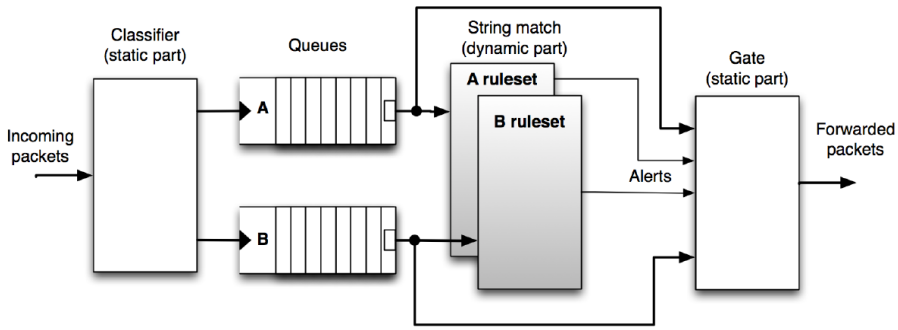


Figure 5.16: Example of a network intrusion detection system that exploits dynamic reconfiguration [76].

protocol. The hardware that processes the different types of packets is time-multiplexed on the same reconfigurable region to save area. Since the reconfigurable region is off-line during reconfiguration, queues are needed to buffer the network traffic and place the packets in the appropriate queue. This part that contains the queues and the sorting logic is placed in the static part of the FPGA since it has to remain active at all times to not drop any packets. The size of the queues is very much dependent on the reconfiguration time. The techniques in this chapter can reduce the amount of memory needed for these queues, making such a system more realistic.

The functional density measure assumes the circuits run at maximum clock frequency. Since the techniques presented in this chapter consider a joint optimization approach they affect the maximum clock frequency at which the circuits can operate. If the circuits have to operate at their maximum clock frequency, it will not be possible to use these techniques. An application in for example High-Performance Computing is therefore unlikely. This is however not always the case. Some applications are more restricted by bandwidth than by pure computing power. The example application in [76] for example runs at a standard clock frequency of 125 MHz. Also, since FPGAs are highly parallel computing platforms they tend to rely more on parallelism than on maximum operating clock frequency.

Of course, as only limited-context DPR is considered in this chapter, the developed techniques are only viable when the number of configurations is limited to no more than 4, as is this case for the network intrusion detection example discussed in this section.

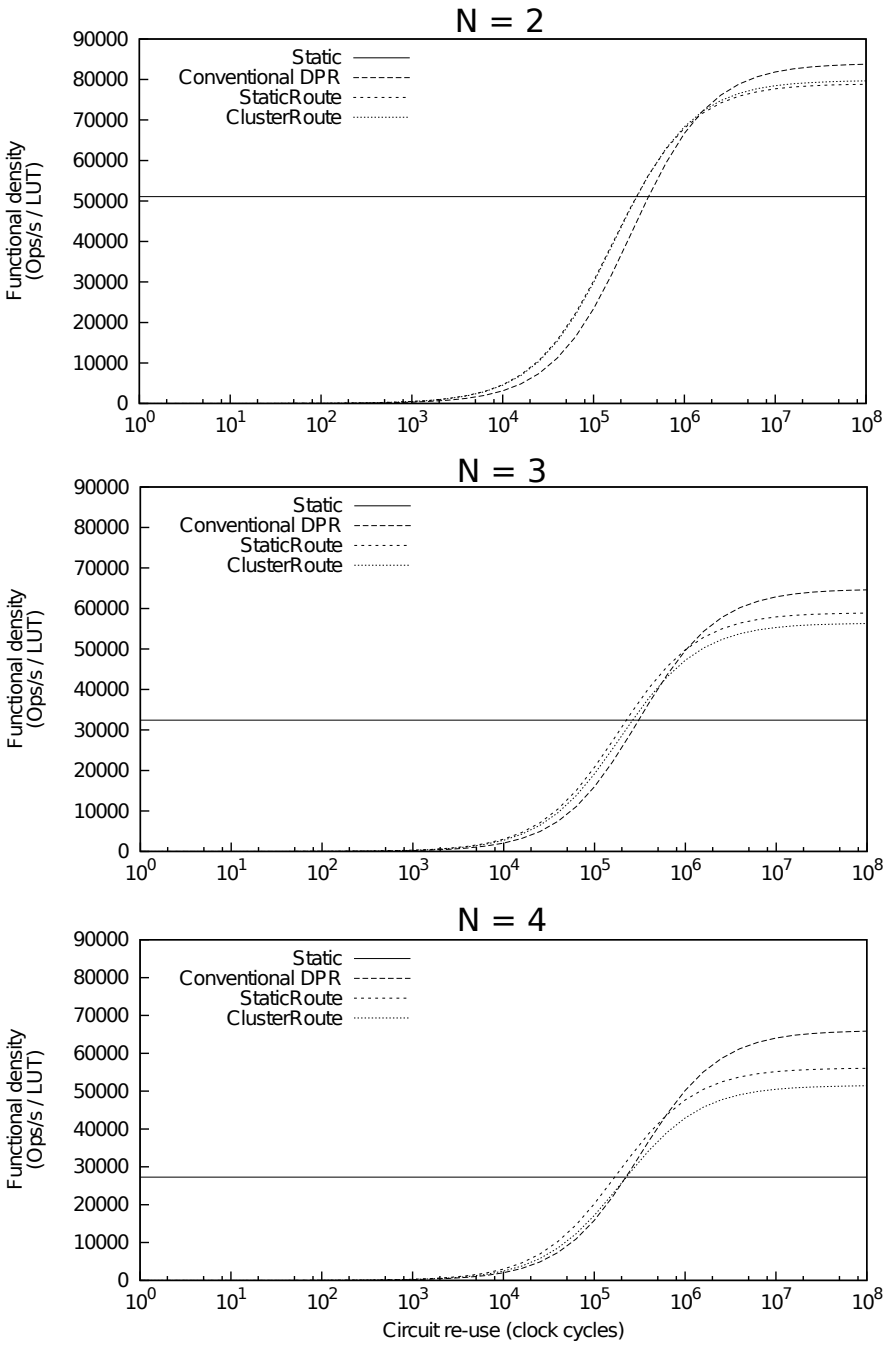


Figure 5.17: Comparison of the functional density of the different approaches.

Chapter 6

An introduction to SRL reconfiguration

6.1 The Shift Register LUT

As was mentioned in Chapter 1, the basic building block of some Xilinx FPGAs is not a simple lookup table (LUT) but a Shift Register LUT (SRL). A schematic representation of an SRL is shown in Figure 6.1. In an SRL the truth table configuration bits are also arranged as a shift register of which the input and the output are accessible from the configurable routing. Therefore these configuration bits are not only accessible through the FPGA's configuration ports (e.g. the Internal Configuration Access Port or ICAP), but also through the

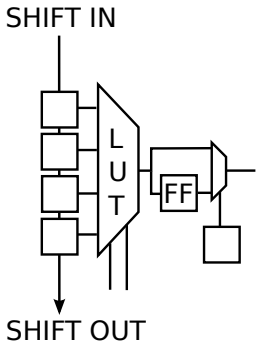


Figure 6.1: A schematic representation of the shift register LUT.

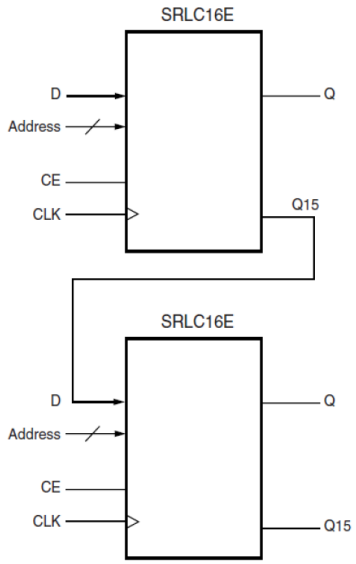


Figure 6.2: A schematic representation of a 32-bit shift register made up out of 2 SRLC16Es.

shift inputs of the SRLs. The SRL provides a very efficient means for implementing shift registers. Instead of using flip-flops of the FPGA fabric, the flip-flops of the configuration memory can be used. As is shown in previous work of other authors, SRLs can also be used to reconfigure the truth table contents of a LUT during run-time [38, 100, 56, 84], hence the term SRL reconfiguration.

In my research on SRL reconfiguration I mainly worked with the Virtex2 Pro. A library component to instantiate SRLs in VHDL for the Virtex 2 Pro is available. Because the Virtex 2 Pro contains 4-LUTs, the shift register that can be made with one SRL has a size of 16 bits, the component is therefore called an SRL16. There are different types available. In my work I mainly used the SRLC16E [101]. The C stands for cascade, and refers to the fact that several of these SRLs can be cascaded to form longer shift registers. The E denotes the presence of a clock enable signal. Figure 6.2 shows a 32-bit shift register made up out of two SRLC16Es. The input port D acts as an shift input to the shift register. Q is the normal output of the LUT. Depending on the address and the content of the truth table, different bit values can be brought out on Q. CE is the clock enable. CLK is the clock

input. When CE is high the bits in the register will shift further one place every clock tick. The signal on D is shifted into the first bit and the last bit is shifted out to the Q15 output. The Q15 output can be connected to the D input of another SRLC16E to create longer shift registers.

6.2 Comparison with ICAP reconfiguration

In this section the intrinsic differences between SRL and ICAP reconfiguration are explained. As will be discussed later, some hardware is needed to interface between the configuration manager and the reconfiguration mechanism. In the case of ICAP reconfiguration the HWICAP, provided by Xilinx, is used for this purpose. For SRL reconfiguration a new interface is developed called the HWSRL. The HWSRL uses, for the Virtex 2 Pro for example, around 140 LUTs and 150 FFs. The hardware resources used are very similar to the HWICAP and are negligible, since the Virtex 2 Pro chip used in this work contains 30000 LUTs. That is why it is not discussed further below.

6.2.1 Accessible configuration bits

When using SRLs only LUTs can be reconfigured. The most important limitation therefore is that the configurable interconnection network of the FPGA can not be changed during run-time using SRLs. With the ICAP on the other hand the complete FPGA fabric can be reconfigured. This is not only the LUTs and the routing, but also specialised hardware components such as hard-wired multipliers.

In Table 6.1 some basic characteristics are listed of the Virtex 2 Pro, Virtex 4 and Virtex 5 FPGAs [44] [103] [107] [102]. It can be seen that in the Virtex 2 Pro every LUT can be set as an SRL. This is however no longer the case for later FPGAs, such as the Virtex 4 and the Virtex 5. Only a fraction of the available LUTs has shift register functionality, while these SRLs are uniformly spread over the FPGA. Also, on the 6-LUT based FPGAs a LUT can only be set as a 32-bit shift register. The more recent Virtex 6 and 7 series FPGAs have the same SRL characteristics as the Virtex 5 [108] [109]. For UltraScale FPGAs, the SRL characteristics are dependent on the type of FPGA [110]. Depending on the type of Ultrascale FPGA 25% or 50% of the LUTs have SRL functionality.

Table 6.1: An overview of the basic characteristics of the Virtex 2 Pro, Virtex 4 and Virtex 5 FPGAs.

Family	K-LUT	LUTs per slice	Slices per CLB	Fraction	Size of SRL
Virtex 2 Pro	4-LUT	2	4	100%	16 bit
Virtex 4	4-LUT	2	4	50%	16 bit
Virtex 5	6-LUT	4	2	25%	32 bit

When using the HWICAP the LUTs that need reconfiguration can be placed anywhere on the FPGA. When using SRLs these LUTs can only be placed where the shift register functionality is available.

6.2.2 Routing cost

When using SRL reconfiguration some routing is needed to implement the reconfiguration paths that interconnect SRLs. Because extra routing is introduced the maximum clock frequency can be affected of the design to be reconfigured. ICAP reconfiguration does not have this cost.

An important remark to this extent is that between SRLs in a Configurable Logic Block (CLB) there is dedicated routing. This routing can only be used to interconnect two consecutive SRLs, it can not be used for other interconnections. With one CLB a 128-bit shift register can be made, that interconnects eight SRLs without any extra routing. Figure 6.3 shows half a CLB or two slices can connect four SRLs without any extra routing. Between the CLBs the normal configurable routing network of the FPGA is used.

6.2.3 Granularity

Every single SRL on the Virtex 2 Pro can be used separately. This allows the creation of random reconfiguration paths on the FPGA. It is, for example, possible to connect two non-consecutive SRLs in a CLB, as is illustrated in Figure 6.4. In the figure the top half of a CLB is shown, in which the first and the fourth SRL are interconnected. Note that in this case it is not possible to use the dedicated rout-

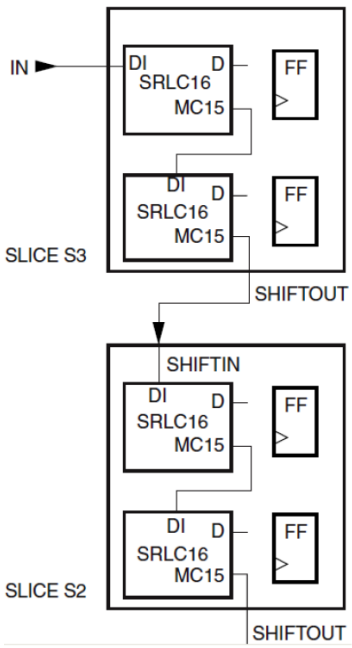


Figure 6.3: Two slices can interconnect 4 SRLs using only the dedicated routing.

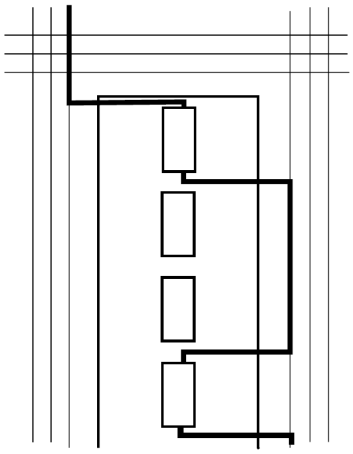


Figure 6.4: Example of a reconfiguration path connecting two non-consecutive SRLs in a CLB.

ing, but instead wires of the configurable interconnection network are used.

The FPGA configuration memory of recent Xilinx devices is arranged as frames that are tiled on the device. A frame is the smallest addressable segment of the configuration memory. When using the ICAP, all operations must therefore act upon complete configuration frames. In the case of the Virtex 2 Pro, for example, the 640 LUTs of a CLB column are spread over two configuration frames of 320 LUTs each. It is therefore not possible to reconfigure one single LUT. The whole frame of 320 LUTs needs to be rewritten. In the best case these are all LUTs that need reconfiguration. Mostly, however, this is not the case and only the truth table content of a few LUTs needs to be changed. It is clear that this is the main disadvantage when working with the ICAP: the reconfiguration time is largely dependent on the number of configuration frames in which the LUTs are placed. This is not the case for SRL reconfiguration.

6.2.4 Reconfiguration bandwidth

In Figure 6.5 a schematic is shown of the top half of one slice of the Virtex 2 Pro. All relevant signals are marked in bold. In this schematic it can be seen that the flip-flop and the SRL in the slice share the same clock. A different clock could be put on the reconfig-

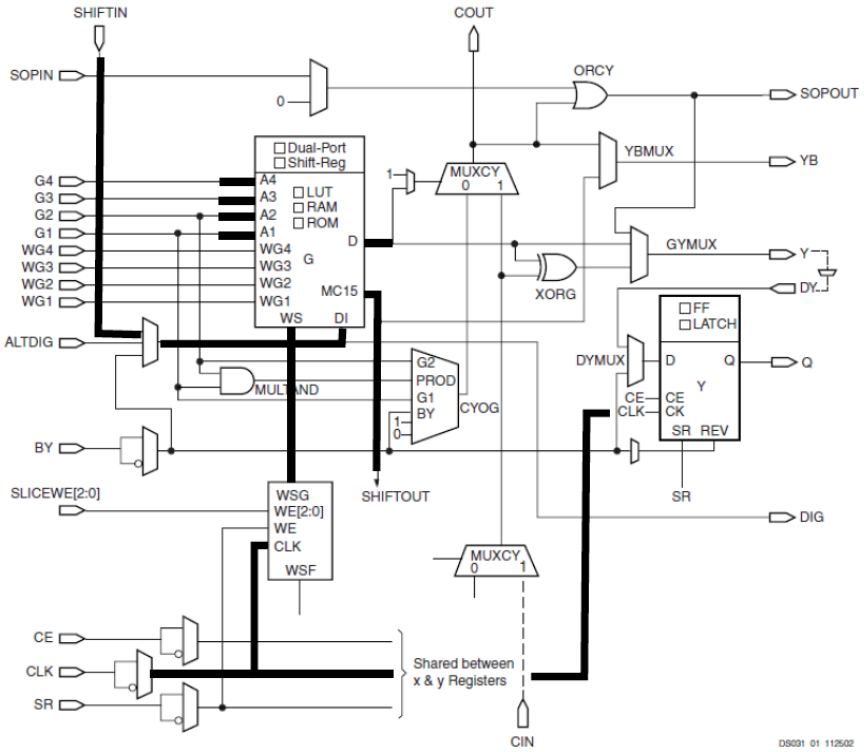


Figure 6.5: A detailed schematic of the top half of a slice on the Virtex 2 Pro.

uration path and possibly use a much higher clock frequency. However, then no flip-flops of the design can be placed in slices that contain SRLs. This could affect the maximum operating clock frequency of the design. In my research this method was used. Further on, it is shown that the impact on the maximum clock frequency of the design is limited, when using the appropriate timing constraints.

Because the interval in which the design operates and the interval in which the bits are shifted in during reconfiguration can not overlap, there is also the possibility to use a clock multiplexer. This last option, however, is associated with a very high design effort.

During my experiments with SRL reconfiguration on the Virtex 2 Pro I found that the clock frequency of the shift registers easily reached a range of 100 to 200 MHz. This could be expected as the reconfiguration paths are very simple. A reconfiguration path is ba-

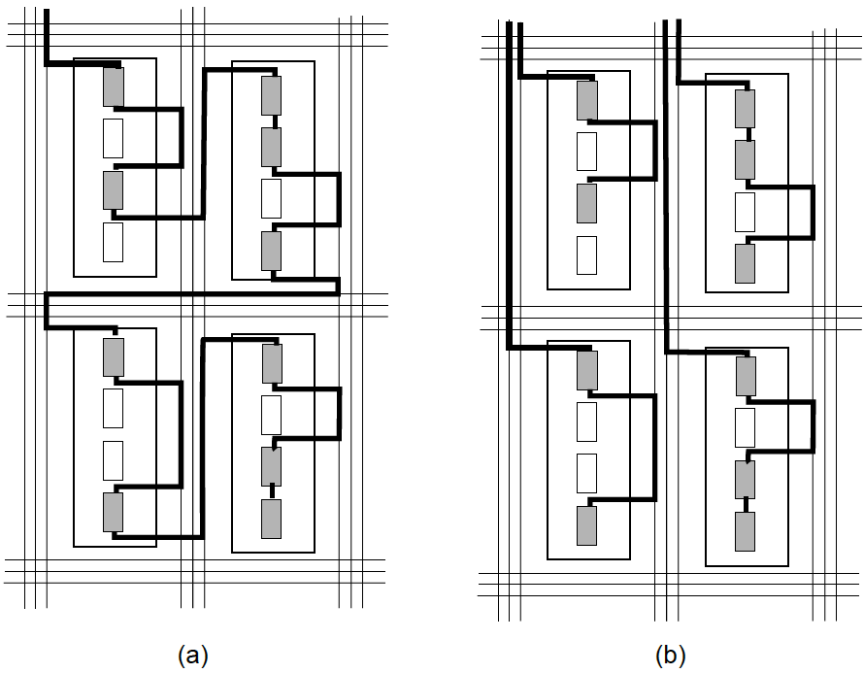


Figure 6.6: An example scenario of SRL reconfiguration with 1 path (a) and with 4 paths (b).

sically a collection of serially interconnected flip-flops. There are no combinatorial paths present, as is the case in the reconfigured design.

The reconfiguration time when using SRLs is not only dependent on the clock frequency at which the bits are shifted in, also the number of bits shifted in is important. This is equal to the number of reconfiguration paths. In Figure 6.6(a) a scenario is depicted in which one reconfiguration path is used to change the truth table contents of the grey LUTs. In Figure 6.6(b) there are four reconfiguration paths shifting in bits in parallel. The designer can therefore tailor the number of reconfiguration paths, and thus also the reconfiguration bandwidth, to the needs of the application.

In the case of ICAP reconfiguration the reconfiguration bandwidth is fixed. The HWICAP of the Virtex 2 Pro, for example, always works at a clock frequency of 66 MHz. The ICAP reconfiguration port on this chip has a width of eight bits.

The differences between SRL and ICAP reconfiguration are sum-

Table 6.2: An overview of the comparison between SRL and ICAP reconfiguration.

	ICAP	SRL
Accessibility	Complete	Only LUTs
Routing cost	None	Extra routing
		minus dedicated routing
Granularity	Frame	LUT
Bandwidth	Fixed	Adaptable

marized in Table 6.2.

6.3 Related work

In this section I give an overview of the literature on dynamic reconfiguration using SRLs.

6.3.1 Dynamic multiplication with a constant

A first application where SRLs are used for dynamic reconfiguration is introduced by FPGA manufacturer Xilinx itself. It is used to implement a dynamically reconfigurable multiplication with a constant [100]. Constant multipliers are multipliers in which one of the factors is a constant. These can be more efficient than generic multipliers in applications where one of the multiplication factors stays constant for a relatively long time interval. In this application from Xilinx, the LUT contents associated with a constant are calculated beforehand. During run-time these truth table values are looked up in a memory and shifted in using the SRLs. The LUTs are all reconfigured in parallel and therefore the reconfiguration process takes 16 clock cycles.

6.3.2 Dynamically reconfigurable PLAs on FPGAs

A second application is found in [84] and [83]. This author implemented Programmable Logic Arrays (PLAs) using LUTs and multiplexers. The most important observation made in this work is that in

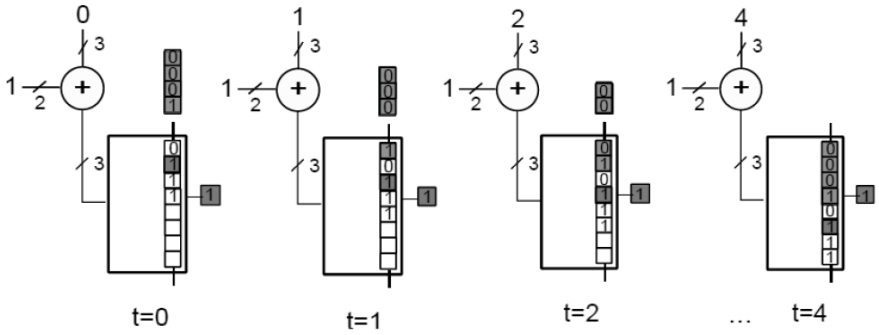


Figure 6.7: An example of SRL reconfiguration that avoids halting the design during reconfiguration.

a PLA the routing is fixed and thus SRLs can be used to reconfigure the LUTs. Using this method, fast dynamic reconfiguration can be achieved. Also, different methods are presented to generate a PLA given the logical circuit of an application.

Note that in the TLUT tool flow the routing is also fixed. The TLUT tool flow uses the conventional FPGA tools and only an adjusted technology mapper. This technique is more powerful than PLAs as composition of Boolean functions is possible. The TLUT tool flow is also more generic and tailored to dynamic reconfiguration. In [84] and [83] the author focuses mainly on the design of the PLA. How this is supposed to be used for dynamic reconfiguration is not discussed.

6.3.3 Dynamic reconfiguration without a transient

Certain specific functions needed in a Universal Mobile Telecommunications System (UMTS) are typically space multiplexed, so that they operate much faster than actually is necessary. In [55] these functions are time-multiplexed on the same FPGA area, increasing the area efficiency of the implementation. Because the author is able to achieve this without lowering performance, this results in a more efficient use of hardware resources.

The shift register functionality of SRLs is used in a special way. This is represented graphically in Figure 6.7. Using this technique, the transient interval in which the design needs to be halted for reconfiguration is avoided.

In the example in Figure 6.7 it seems that 3-LUTs are used, but these 3-LUTs are actually used as 2 2-LUTs. The truth table of a 2-LUT contains four bits, that of a 3-LUT eight bits, so the 3-LUT contains two 2-LUTs. Reconfiguration without a transient is achieved by letting the input of the 3-LUT follow the top 4-bit truth table as it is shifted down. A second 4-bit truth table can therefore be shifted in without disturbing the designs functioning. This illustrated in the figure, because the output remains the same, while a new top 2-LUT is shifted in. To make the input follow the truth table an adder is used. In this example the actual switching between truth tables occurs for $t > 4$, when the number that is added with the input is set to zero again. The contents of the truth tables are generated on-line and shifted in immediately. The advantage of this approach is that no extra memory is needed.

In [55] the author uses an architecture with 4-LUTs and maps the reconfigured module to 3-LUTs. Exactly the same principle applies as in our example with 3-LUTs. Note that, in principal an adder is needed for every LUT that needs reconfiguration. This technique is thus especially useful when many of these LUTs share inputs. For the application under consideration only one adder is needed as the input to the reconfigurable module is also the input to all the 3-LUTs it consists of.

This technique is very interesting for designs in which the new truth table contents are known before the reconfiguration has to take place. This way computation time and reconfiguration time can be overlapped.

Note that the same result could be achieved as in this work using the ICAP, since recent FPGAs allow for glitch-less reconfiguration: when a memory cell in the configuration memory is overwritten with the same value, it is as if this value did not change [85]. Hence the design can operate during reconfiguration using the ICAP, as long as the active truth table contents are overwritten with the same value. In this case the truth table bits are not shifted, but overwritten. Only one half of the truth table would be active at any time. The other half could be reconfigured without interrupting the design's operation. The most significant bit of the 4-LUTs would in this case be used to switch between the two halves. This way the adders that are needed in [55] are avoided and it is also no longer needed that the 3-LUTs should share inputs. The reconfiguration will take longer than is the case with SRLs, which only needs eight clock ticks. The time interval

between reconfigurations should therefore be longer.

6.3.4 Efficient implementation of low frequency finite state machines

In [56] the observation is made that some finite state machines (FSMs) only change state once in a while. It therefore makes sense to specialise the hardware for the current state. A method is explained that makes it possible to implement an FSM with less LUTs using dynamic reconfiguration. Although the author proposes a new architecture for this purpose, the design of such an FSM still happens manually.

Since FSMs only implement a limited number of states, it is possible to generate the LUT contents off-line and save these to BRAM. So this system trades off some of the gains made with more BRAM usage. According to this work, however, it is likely that there are some spare BRAM resources available.

The routing in the design is fixed because the reconfigured logic consists of a large lookup table built up using LUTs. SRL reconfiguration can therefore be used to shift in the data out of the BRAM into the LUTs. Reconfiguration takes 16 clock cycles, since a 4-LUT architecture is used and all the configuration bits are shifted in in parallel.

6.3.5 Regular expression matching

A custom designed architecture for regular expression matching is proposed in [38]. The shift register functionality is used to reconfigure the LUTs of specific, hand-designed functions. The authors present a tool that automatically generates VHDL code that can match a certain regular expression. It is not possible to match a regular expression for which the VHDL code was not already generated off-line. The configuration data is also generated off-line by running the conventional FPGA tool flow. During run-time, the configuration data is looked up in a memory and shifted in accordingly.

6.3.6 Discussion

What stands out in the applications of SRL reconfiguration above is that almost all the applications are manually designed. The work in [84] and [83] allows for generic design, but uses the FPGA suboptimally since for any application a PLA is generated, which is es-

sentially a two-level Boolean function. It is also not explained how this can be used for dynamic reconfiguration. In most cases considered, the configuration data is generated off-line and shifted in during run-time. This is not a general approach, since some applications would require an enormous amount of memory.

Most authors use the SRLs to reconfigure small, specific, hand-designed functions. It is not the case that the LUTs that need to be reconfigured are spread out in the design. To the best of my knowledge, before I started my research no work was done that attempts to figure out the influence the SRL reconfiguration paths have on the design. What is, for example, the impact on the maximum operating clock frequency of the design? What is the expected speed up of the reconfiguration process that can be obtained with SRL reconfiguration? Does the order in which the LUTs are placed in the reconfiguration paths matter? I will investigate these questions in Chapter 7.

Chapter 7

SRL reconfiguration of regularly structured designs

The TLUT toolflow was presented in Section 3.2.1. I also discussed how shift register LUTs have been used for dynamic reconfiguration in previous work in Section 6.3. SRLs have been primarily used to reconfigure specific hand-designed functions. In this chapter, I explain how I integrated SRL reconfiguration in the automatic TLUT tool flow.

In regularly structured designs the same hardware module is instantiated many times. For these designs, I quantify the speed-up of the reconfiguration process and also show that the memory efficiency thereof can be greatly improved using SRLs. The impact of SRL reconfiguration on the maximum operating clock frequency of the design is also discussed.

7.1 Integration of SRL reconfiguration in the TLUT tool flow

The original TLUT tool flow that generates a master configuration and a reconfiguration procedure is described earlier in Section 3.2.1. The tool flow that incorporates SRL reconfiguration is depicted in Figure 7.1. Adjustments have been made in three different steps of the tool flow. First, a step has been added in which the SRL reconfiguration paths are introduced. Second, during the placement and routing step, appropriate timing constraints are necessary for the proper optimization of the design clock and shift register clock. Finally, in-

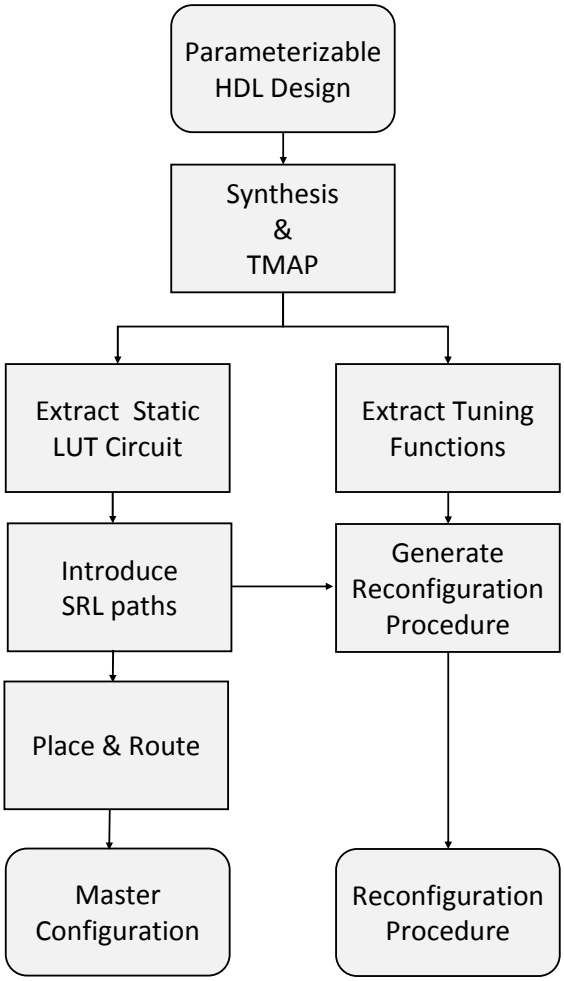


Figure 7.1: The TLUT tool flow with SRL reconfiguration.

stead of using the HWICAP, a newly designed block is used called the HWSRL. This new block receives the configuration data and is responsible for shifting it in the reconfiguration paths. The reconfiguration procedure is adapted to reflect this change. Now, the address of the LUTs in the configuration memory is no longer necessary. The only relevant information is how the LUTs are arranged in the reconfiguration paths.

7.1.1 Introduction of the reconfiguration paths

In the VHDL files generated by the SRL-based TLUT tool flow of Figure 7.1, SRL16 primitives are instantiated instead of regular LUTs. By default only one reconfiguration path is used. If the TLUT circuit is completely contained in one VHDL file, the generation of the reconfiguration path is completely automatic. The SRLs are placed in the reconfiguration paths in the order of appearance in the VHDL file. When there are multiple VHDL blocks, or instantiations of blocks, the tool flow will automatically generate an adapted version for each VHDL file. In these VHDL files shift in and shift out ports are added and the SRLs are also interconnected in the order of appearance. However, the designer has to manually interconnect these different blocks. Creating a completely automatic method on VHDL level is possible, but would require an extensive VHDL parser on system-level. Creating such a parser is not trivial and very time-consuming.

Another, more generic way of inserting the SRLs, was presented in [45]. This new tool flow was developed by Karel Heyse and based on the VHDL-based tool flow presented in this chapter. In this new method, the reconfiguration paths are inserted after placement on the complete, flattened out design. The SRLs are inserted in an intermediate representation of the design offered by Xilinx, which is called the Xilinx Design Language (XDL). A small excerpt of an XDL file after technology mapping is shown in Figure 7.2, together with the slice it relates to. Xilinx also provides access to the implementation of a design through the XDL interface after placement and routing. The advantage of inserting the reconfiguration paths in XDL is that there is more control on how these are implemented. When these are inserted in VHDL the Xilinx tools determine the exact implementation.

In Figure 7.3 it is illustrated how the reconfiguration path is inserted in the VHDL-based flow for regularly structured designs, in

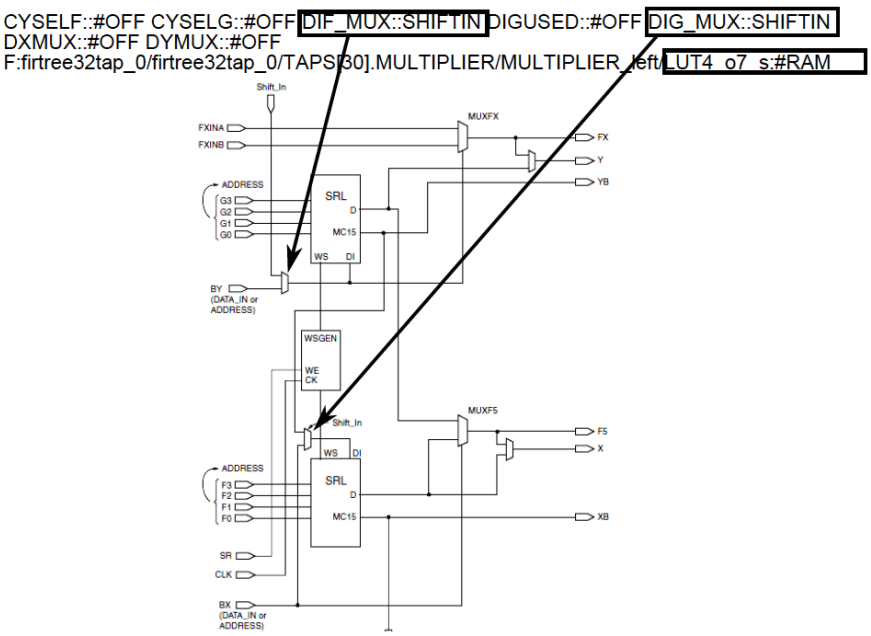


Figure 7.2: A small excerpt out of a XDL file together with the slice it relates to.

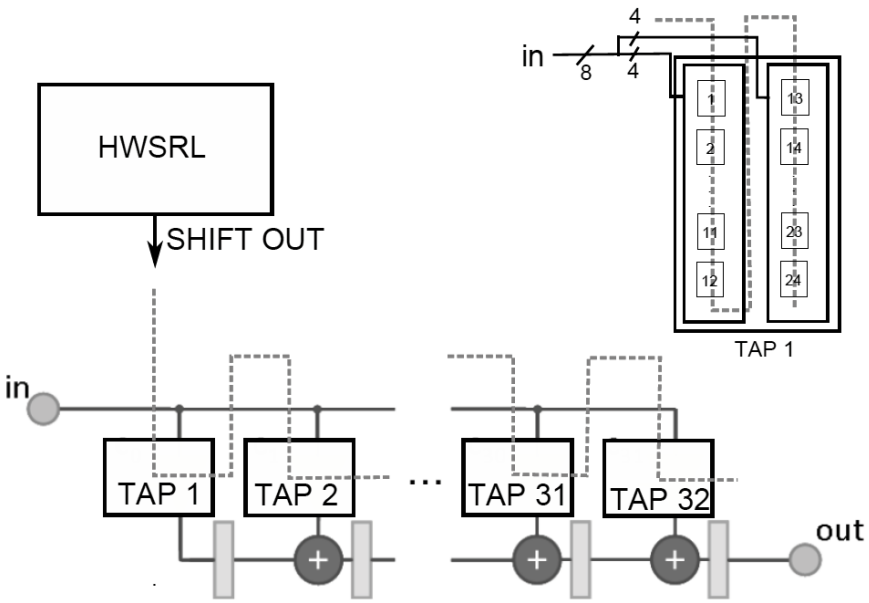


Figure 7.3: The SRL reconfiguration path in a regularly structured adaptive filtering application.

this case a 32-tap adaptive filter. The paths basically follow the regular structure. The input of the reconfiguration path is connected to the HWSRL block. The different taps, which are 8 by 8 multipliers of the FIR filter, are instantiated using a generate statement. The reconfiguration path is easily inserted in VHDL by incorporating the adapted VHDL file that contains a shift in and a shift out port in the generate statement. Each 8 by 8 by 8 multiplier consists of two 4 by 8 multipliers. Also a connection has to be made between these 4 by 8 multipliers.

7.1.2 Adding appropriate timing constraints

After synthesis an estimation is made of a theoretical maximum clock frequency based on the gate delays of the primitives present in the synthesized circuit. The primitives are for example LUTs and carry chains. This maximum clock frequency is an over estimation, since it will be reduced by the effect of technology mapping, placement and routing, but already gives an indication of the expected maximum clock frequency.

When analysing the synthesis tool reports it is notable that the estimated maximum achievable clock frequency of the FIR filter design with reconfiguration paths is lower than the original design without reconfiguration paths. On the Virtex 2 Pro the synthesis tool reports a maximum clock frequency of around 165 MHz for the original design and 135 MHz for the design with reconfiguration paths. This is striking since it could be expected that the reconfiguration paths would only have effect during placement and routing.

When the synthesis report is analysed in more detail, it can be seen that the delay of a SRLC16E primitive is larger than that of a LUT. This is illustrated in Figure 7.4. It is this delay that results in the longest path when estimating the maximum clock frequency of the design. This observation can be verified by instantiating the SRLC16E components, but not adding the reconfiguration paths. The synthesis report still indicates a lower clock frequency for the design with SRLs. The reduction in clock frequency is therefore not due to the reconfiguration paths, but the SRLC16E components themselves.

This is an interesting finding, since it could be expected that the reduction in performance of the design would primarily come from the addition of the reconfiguration paths. However, the SRLC16E component itself seems to also have an influence. The reason for

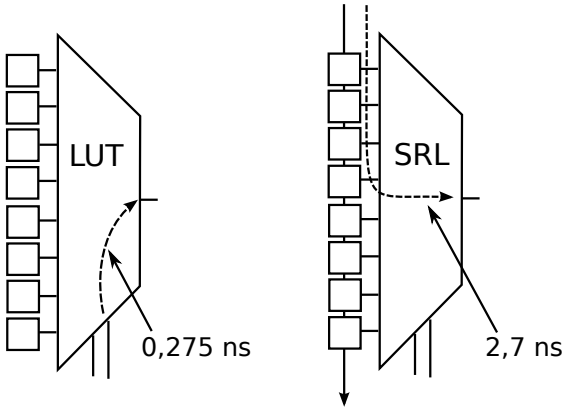


Figure 7.4: The gate delay of a shift register LUT compared to that of a regular LUT.

this is that the tools treat the shift register LUTs as shift registers. The tools assume that there could be data shifted in the SRLs during operation. That is why the tools assign a higher delay than normal LUTs to the SRLs. The truth table bits of LUTs are indeed assumed not to change during operation.

In the case of SRL reconfiguration, however, the truth table contents also do not change during operation of the design. It is only during the reconfiguration process, when the design is halted, that the new truth table contents are shifted in. Therefore, in our case the delay of an SRLC16E is the same as that of a LUT. For a proper optimization of the clocks, it is important that this is made clear to the tools.

A possible solution is to put the reconfiguration paths on a different clock than the design and tell the tools to ignore the transitions between the two clock domains. More specifically this is done by putting the SRLs and the FFs in a different timing group and ignoring the transitions between these groups. This is the approach taken in my work. When applying this solution, there is another, much smaller, side effect to be considered. The SRLC16Es and the FFs of the design can not be placed in the same slice. As I explained in Section 6.2.4 the SRL and the FF of a slice share the same clock.

Another observation was made when I put a clock frequency constraint of 200 MHz on the reconfiguration paths and a clock of 100

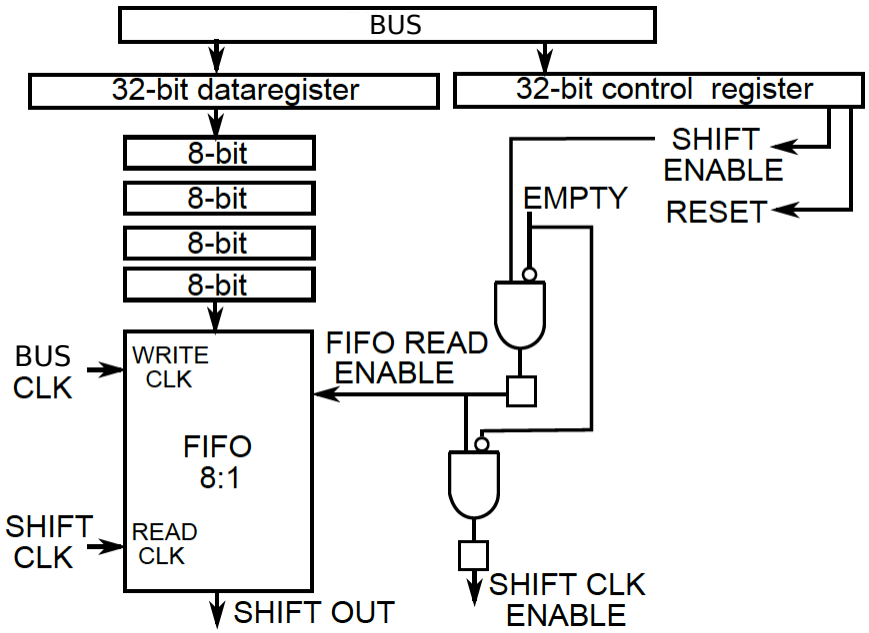


Figure 7.5: Design of the HWSRL block.

MHz on the design. When running the placement and routing tools as is, they failed to reach both constraints. However, if the design clock is given a priority of 1 (highest) and the reconfiguration clock a priority of 100, the tools easily reach the given constraints.

It seems that the tools get confused when trying to optimize globally for both clock domains. This could be due to the overestimation of the importance of the reconfiguration clock. The reconfiguration paths are very simple and consist solely out of FFs with interconnection between the SRLs. By giving the reconfiguration clock a very low priority, the tools are basically told to ignore the reconfiguration clocks. As will be discussed further on, even when the tools only optimize the design's clock, the reconfiguration clock still easily reaches a high clock frequency.

7.1.3 The HWSRL block

The design of the proof-of-concept implementation of the HWSRL block in the case of the Virtex 2 Pro is shown schematically in Figure 7.5.

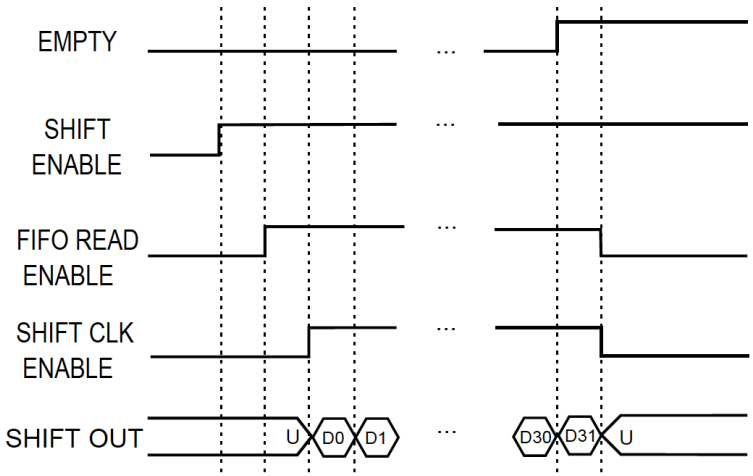


Figure 7.6: A timing diagram of the control signals of the HWSRL block.

There is a 32-bit data register and a 32-bit control register available, which are connected to a bus. The Peripheral Local Bus (PLB) is used. This bus can be written from the C code in the reconfiguration procedure. The HWSRL block contains a FIFO which is used for synchronization between the PLB and the logic that shifts in the configuration data. Writing and reading the FIFO can happen using different clocks. The FIFO can also be used to temporarily buffer configuration data on the FPGA.

The read width of the FIFO is set to the number of reconfiguration paths. In our case this is set to 1, by default. This simplifies the hardware needed to shift in the configuration data, because the data can flow out of the FIFO without interruption. No back-end is necessary to read the data out of the FIFO and write it to the reconfiguration paths. The output of the FIFO is therefore directly connected to the first SRL of the reconfiguration path.

The FIFO is generated and optimized by the software tools of Xilinx. This software does not allow a ratio larger than 8 between the read and write widths. That is why the write width is 8. Therefore a front-end is required to sequentially write the bits out of the data register into the FIFO.

A timing diagram of the control signals responsible for shifting in the configuration data is shown in Figure 7.6. Shifting the data

out of the FIFO is controlled by the SHIFT ENABLE signal. As can be seen in the figure, when this signal is put high 32 bits are shifted out of the FIFO and in the reconfiguration path. The READ ENABLE signal follows the SHIFT ENABLE signal, unless the FIFO is empty (EMPTY signal high), then this signal stays low. When the READ ENABLE signal is set high, the 32 bits will be present on the output of the FIFO. In Section 6.1, it was explained that data could be shifted in the SRLC16E by putting the CE (clock enable) signal high. The signal that drives all the CE signals of the SRLs is called SHIFT CLK ENABLE. This signal follows the READ ENABLE signal, unless the EMPTY signal is high, then it stays low. This last condition is necessary, because the READ ENABLE signal stays high one clock tick too long when the FIFO becomes empty. When the EMPTY signal turns high, there is no data in the FIFO any more, yet the READ ENABLE signal stays high for one more clock cycle. Note that no counter is used in the implementation of the HWSRL block, instead the EMPTY signal is used. That is why only configuration data is allowed to be sent to the FIFO.

In the design a RESET signal was added that can set all the control signals to zero and reset the FIFO. The implementation of the HWSRL block on the Virtex 2 Pro uses 137 LUTs and 156 FFs, which is very similar to the HWICAP and is negligible.

7.2 Reconfiguration speed up

7.2.1 Experimental set-up

The adapted TLUT tool flow is validated on an adaptive filtering application: a fully pipelined 32-taps FIR filter with 8-bit coefficients and an 8-bit input. It is assumed that the coefficients need to be changed every once in a while, which could, e.g., be the case in a wifi application to cancel inter-symbol interference (ISI). Every time a wifi-client is moved, the communication channel properties change and the coefficients of the ISI cancelation filter need to be updated. It is also assumed that the configuration manager (in our case the PowerPC) is responsible for calculating the new coefficients and reconfiguring the filter accordingly.

I implemented both platforms of Figure 7.7 on a Xilinx XUP board, that contains a Virtex 2 Pro (XC2VP30) FPGA. The first implementation of the reconfigurable FIR filter uses our previous ICAP-

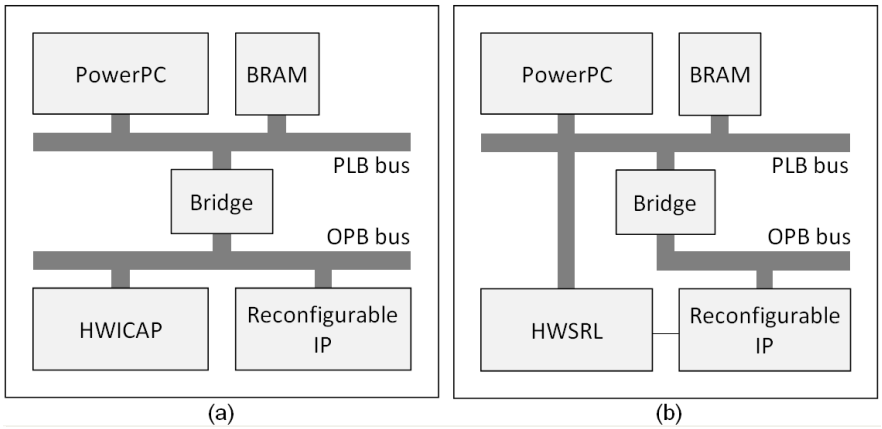


Figure 7.7: Platform implementation of the original TLUT tool flow (a) and the new SRL implementation (b).

based tool flow [19]. The configuration manager is implemented in software and changes the filter characteristics by reconfiguring the FPGA through the ICAP. The second implementation uses our new SRL-based tool flow. The configuration manager is again implemented in software, but now reconfigures the filter by using SRL reconfiguration. The functionality of the 32-taps FIR filter is also tested in both implementations by comparing it to a FIR filter simulated in software.

As shown in [19], the TLUT tool flow implements the FIR-filter example with an area gain of 40 percent compared to a generic design without RTR. The same gain is found in the SRL-based approach. There is a large difference in the overhead though. While the ICAP-based platform requires 141 ms (taken from [19]) for the reconfiguration of the TLUTs, our new SRL-based implementation needs less than 1 ms, over two orders of magnitude improvement. However, as the reported reconfiguration time for the ICAP solution is mostly due to the overhead of the specific implementation, I further explore the intrinsic differences in reconfiguration time, and its scalability, in the next section.

The overhead comparison of the previous paragraph is not completely fair as the ICAP-based tool flow uses the Xilinx function `XHwIcap_SetClbBits` and reconfigures one LUT at a time. In fact, the

Table 7.1: Results for the HWICAP reconfiguration.

	Maximum clk freq	Number of frames	Reconfiguration time ICAP
32 taps	151 MHz	68	840 μs
64 taps	153 MHz	88	1087 μs
128 taps	147 MHz	131	1619 μs

HWICAP is capable of framewise reconfiguration.[1] To compare the fundamental reconfiguration overhead difference, it is assumed that the reconfiguration data is already available on the FPGA, for example in a RAM block, and that it can be processed in one time. For the ICAP the reconfiguration data is the partial reconfiguration bitstream the HWICAP has to process. In case of the SRL reconfiguration the reconfiguration data consists of the configuration bits of the TLUTs that have to be shifted in by the HWSRL. With this assumption, the complete platform (with PowerPC, HWICAP or HWSRL, ...) are again implemented as shown in Figure 7.7. The scaling behaviour is investigated below by implementing a 32-tap, a 64-tap, and a 128-tap FIR filter. The implementation with the 32-tap FIR filter takes 6% of the FPGA area, for the 128-tap FIR filter this is 40%.

7.2.2 Results

7.2.2.1 Reconfiguration speed-up

In Table 7.1 the measurements for the design that uses ICAP reconfiguration are shown. As the maximum clock frequency is independent of the reconfiguration infrastructure, it is only determined by the reconfigurable module. The number of frames in the partial configuration bit stream naturally increases with the number of taps implemented. The instructions in the partial configuration bit stream of the ICAP are very few and are therefore neglected. The number of frames is used to estimate the reconfiguration time. These numbers also include the padding frames that have to be processed by the

[1] A frame is a column of 1-bit wide in the configuration memory. For the Virtex 2 Pro a frame consists of 204 bytes.

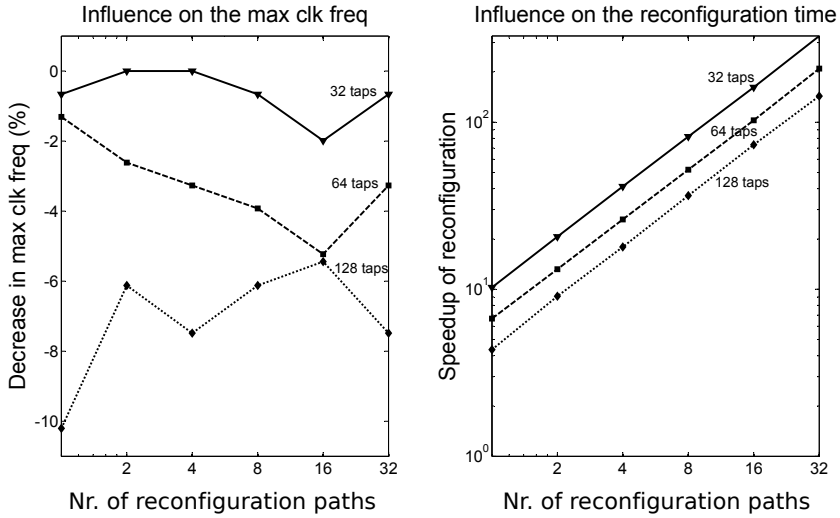


Figure 7.8: (a) Influence of SRL reconfiguration on the maximum clock frequency of different designs. (b) Speedup of the SRL reconfiguration over the ICAP reconfiguration for the different designs. Both figures show the scaling over the number of reconfiguration paths.

ICAP. It is known that a frame of the Virtex 2 Pro consists of 204 bytes and that the HWICAP processes it bytewise at a clock frequency of 66 MHz. This results in very good estimates for the reconfiguration time of the different designs (last column of the table), as the number of instruction bytes is negligible.

In the case of the SRL reconfiguration, the reconfiguration time depends on the number of reconfiguration paths so the FIR filters are reconfigured using 1, 2, 4, 8, 16, or 32 ad hoc reconfiguration paths of equal length. The clock of the reconfiguration paths is put on the same value as the clock of the design. As it is known what the number of LUTs is that has to be reconfigured and the clock frequency at which the bits are shifted in, it can easily be calculated what the reconfiguration time is for the different implementations. Figure 7.8(a) shows the decrease in maximum clock frequency for the different FIR filters as function of the number of reconfiguration paths relative to the values of the design clock without these paths (as mentioned in Table 7.1). Figure 7.8(b) shows the speed-up of the reconfiguration time over the respective ICAP results.

Figure 7.8(b) shows that SRL reconfiguration with 32 reconfiguration paths is two orders of magnitude faster than ICAP reconfiguration. Indeed, the partial configuration bit stream for the ICAP consists for more than a third of padding frames. Moreover, the ICAP bandwidth is fixed. In the case of the SRL reconfiguration, only the LUTs that have to be reconfigured are put in the reconfiguration paths. Increasing the number of reconfiguration paths reduces the number of bits per reconfiguration path and increases the reconfiguration speed. In the SRL case, the reconfiguration speed can thus be tuned to the application requirements.

7.2.2.2 Impact on maximum clock frequency

The decrease in maximum clock frequency of the design for the fastest reconfiguration approach, the one with 32 reconfiguration paths, stays within 8%. The maximum reduction in clock frequency is 10%, for the case of the 128-taps FIR filter reconfigured with 1 path. It can also be observed that the number of reconfiguration paths seems to have little influence on the clock frequency. The largest effect comes from the number of taps of the FIR filter. This could indicate that more routing congestion arises as the number of taps of the FIR filter increases.

These results are confirmed by the experiments done in [45]. Besides an adaptive filtering application (FIR), the authors in this work also considered a Ternary Content Addressable Memory (TCAM). The main results regarding the impact on the maximum clock frequency can be found in Figure 7.9 for the FIR filters and Figure 7.10 for the TCAMs and are discussed in the following paragraphs.

The clock frequency of the method where the reconfiguration paths are inserted following the regular structure of the design are labelled 'Structured SRL chain'. Also here the maximum decrease in maximum clock frequency is around 10% (TCAM design with 128 elements). Choosing the reconfiguration paths as such also outperforms the method proposed in [45], which is labelled 'mTSP SRL chain' on the figures. This method solves a multiple Travelling Salesman Problem based on the implementation of the design after placement. This method is, however, universally applicable and not only useful for regularly structured designs.

In a lot of cases, it can be observed that the maximum clock frequency even improves compared to the design without reconfigura-

tion paths. This has to do with the fact that the SRLs also influence the packing, placement and routing algorithms. In these cases this seems to be a positive influence. The results which are labelled 'With SRL' show that this effect already occurs when the SRLs are added, together with the reconfiguration clock and the shift enable signals, but without the reconfiguration paths.

In my master thesis [3] I showed that regularly structured reconfiguration paths outperformed random paths significantly. This result is also confirmed by the work done in [45], as can be seen in Figure 7.9 and Figure 7.10. The maximum clock frequency is up to around 40% worse than the structured reconfiguration paths (FIR, 256 taps).

Finally, note that all the experiments above were done on the Virtex 2 Pro FPGA architecture. In Section 6.2.1 it was mentioned that in later FPGAs, such as the Virtex 4 FPGA and the Virtex 5 FPGA, not every LUT can be set as an SRL. The fraction of SRLs is 50% on the Virtex 4 FPGA and 25% on the Virtex 5 FPGA. In [45] it is shown that on these later FPGAs the maximum clock frequency deteriorates more than on the Virtex 2 Pro. This effect is most noticeable when the fraction of TLUTs in the design is higher than the fraction of SRLs available.

7.3 Memory efficiency

As was presented earlier, the TLUT toolflow automatically maps an annotated VHDL description onto a TLUT circuit. TLUT circuits are very memory efficient, since the specialized LUT circuits are not stored separately, but as Boolean functions of a parameter. Note that the routing and LUT truth table configuration bits of recent Xilinx Virtex FPGAs reside in different configuration frames [105]. Since the routing of a TLUT circuit is fixed, no information has to be stored in the configuration manager (CM) concerning the routing.

In regularly structured designs the same hardware module is instantiated many times. An example of a regularly structured design is an adaptive filter in which a module is instantiated that performs a multiply-accumulate operation. The function of every module is the same, only the inputs that have to be processed are different. Using a TLUT circuit as a module of regularly structured designs provides a means for memory efficient and fast run-time reconfiguration of such designs. The Partial Parameterized Configuration (PPC) expresses

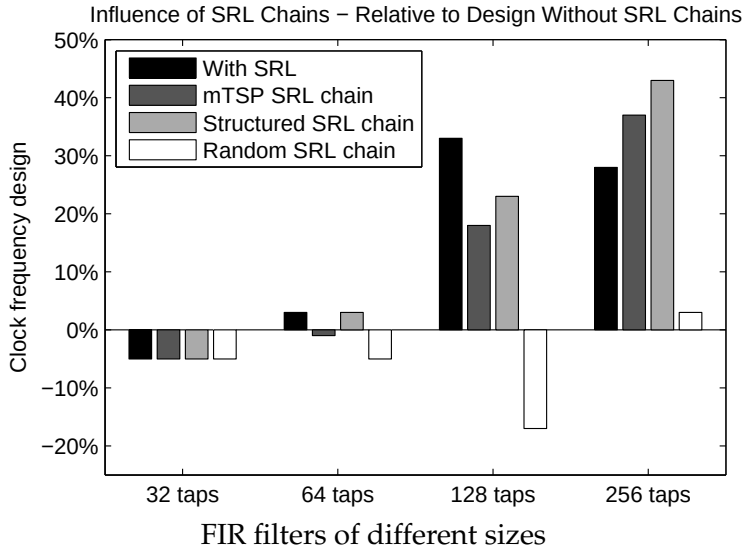


Figure 7.9: Influence of the reconfiguration chains on the FIR filter design's clock frequency using the Xilinx tool flow, averaged over experiments with 1, 4, 16 and 32 chains

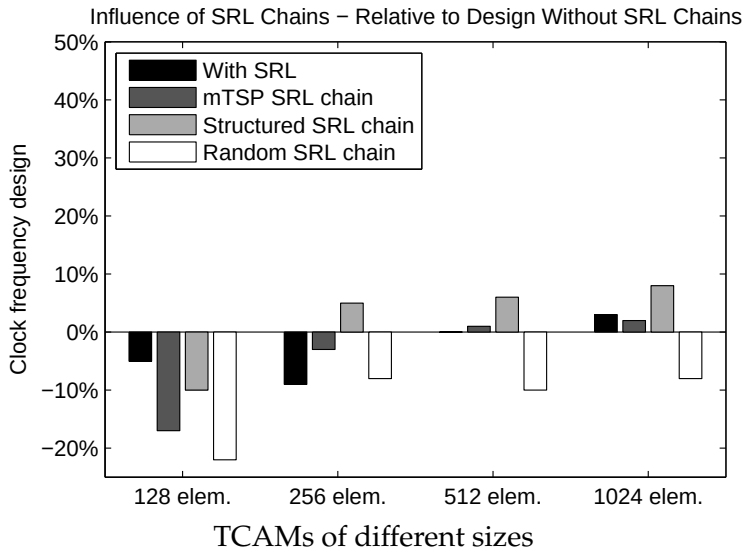


Figure 7.10: Influence of the reconfiguration chains on the TCAM design's clock frequency using the Xilinx tool flow, averaged over experiments with 1, 4, 16 and 32 chains

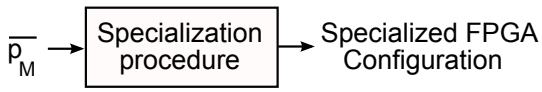


Figure 7.11: The specialization procedure in the case of regularly structured designs.

the truth table contents as a Boolean function of the parameters. This Boolean function is stored as a logic network of AND and inverter gates. Because the function of all the modules is the same, the PPC only has to be stored once in the specialization procedure.

The specialization procedure in the case of regularly structured designs is shown in Figure 7.11. During run-time this procedure takes in a list $\overline{p_M}$ containing the parameter values of the different modules and generates a specialized FPGA configuration. Because the PPC only has to be stored once, the memory requirements for the evaluation of the PPC are constant and independent of the number of modules. However, the memory requirements of the specialization procedure are also dependent on the information needed for reconfiguration. In this section the memory usage of the specialization procedure is compared when using ICAP and SRL reconfiguration. Below it is shown that for ICAP reconfiguration the memory usage is dependent on the number of modules, which results in excessive memory usage when the number of modules is increased. For SRL reconfiguration it is shown that the reconfiguration paths can be constructed in such a way that no information needs to be stored for reconfiguration. Particular interest is shown in how the memory usage, expressed in bits, of the specialization procedure scales with the number of modules. In every section the reconfiguration time is also briefly discussed.

7.3.1 ICAP reconfiguration

The ICAP reconfiguration interface processes the reconfiguration data per frame. The pseudo code for the specialization procedure is in this case given in Figure 7.12. Note that the size of the C code that implements this relatively simple pseudo code will be small and independent of the number of modules. To estimate the memory usage when increasing the number of modules, the size of the actual

```
function specializationICAP(parameterList)
  for frameAddress in frameAddressArray:
    Frame = getConstantData(frameAddress);
    for each Tlut in Frame:
      (frameIndex,tlutPPCFunction,parameterIndex) =
      getLUTInfo(Tlut);
      Frame [frameIndex] =
      evaluateTLUT(tlutPPCFunction, parameterIndex);
    configureFPGA(Frame, frameAddress);
```

Figure 7.12: Pseudo code for ICAP specialization.

specialization procedure is thus neglected and the focus lies on the amount of information needed in the specialization procedure.

The placement process distributes the TLUTs of different modules irregularly across various different configuration frames. This is illustrated in Figure 7.13(a) for a regularly structured design with two modules and three TLUTs per module that are scattered across two frames. This scattering has several consequences.

First of all, information has to be stored containing the locations of the TLUTs in the configuration memory. This location information consists of a frame address and an index in the frame. The number of modules is denoted M, the number of TLUTs per module L_M, the number of frames F in which the TLUTs are located, the number of LUTs per frame L_F and the number of bits to store the frame address b_A. The number of bits needed to store this information is then $F \cdot b_A + M \cdot L_M \lceil log2(L_F) \rceil$.

Second, the TLUTs of one module are also distributed irregularly across several different frames. To accomodate evaluation on a frame basis, as is shown in Figure 7.12, the PPC of the module must be adjusted to accomodate evaluation per TLUT. Per TLUT of the module a separate PPC is thus created. Per TLUT one also has to store the function to be called to evaluate the PPC of the TLUT and a pointer to the parameter value of the respective module in the parameter list $\overline{p_M}$. The main consequence is that Boolean gates of the PPC of the module, that are reused across TLUTs will have to be duplicated. This increases the memory requirements for the compiled C-code that evaluates the PPC. Since the function of all the modules is the same, this C-code also only has to be stored once. For example, all

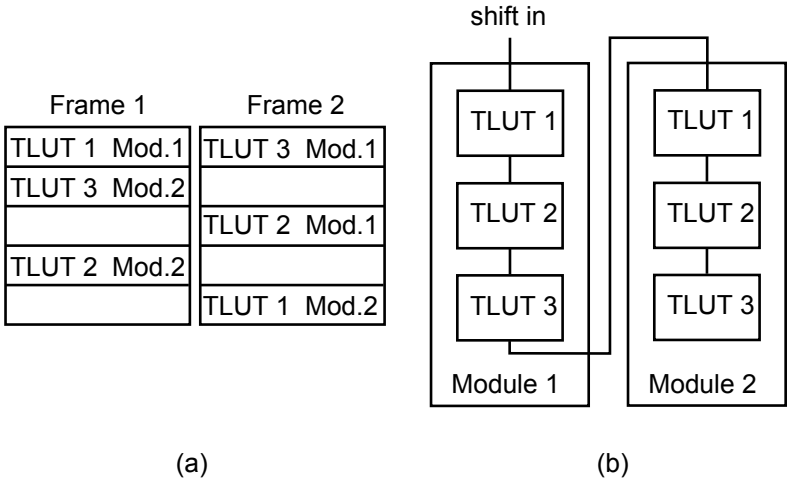


Figure 7.13: The LUTs of a regularly structured design irregularly scattered in the ICAP configuration memory (a) and regularly placed in an SRL reconfiguration path (b).

the TLUTs with name 'TLUT1' in Figure 7.13 will use the same PPC. PPC_{LUT} is denoted as the sum of the sizes of the compiled C-code of the PPCs of the different TLUTs. The number of bits needed to store this information is then $PPC_{LUT}+M\cdot L_M\cdot(\lceil log2(M)\rceil+\lceil log2(L_M)\rceil)$.

Finally, when reconfiguring the TLUTs, the constant data in the frames is also rewritten. One option is to keep this constant data in the memory of the CM. This is very memory consuming, but has the fastest reconfiguration time. Since the configuration memory can also be read a read-modify-write strategy can also be applied, where reconfiguration time is sacrificed for more memory efficiency.

$$M_W = PPC_{LUT} + F \cdot (b_A + b_F) + M \cdot L_M \cdot b_L \tag{7.1}$$

$$M_{RMW} = PPC_{LUT} + F \cdot b_A + M \cdot L_M \cdot b_L \tag{7.2}$$

where b_L is given by $\lceil(log2(L_F))\rceil + \lceil(log2(L_M))\rceil + \lceil(log2(M))\rceil$ and b_F is the number of bits per frame. In both equations (7.1) and (7.2) it can be seen that the memory usage grows with the number of modules.

The reconfiguration time is given for both versions of ICAP reconfiguration, in equations (7.3) and (7.4).

$$T_W = \frac{F \cdot b_F \cdot T_{ICAP}}{D} \tag{7.3}$$

$$T_{RMW} = 2 \cdot T_W \tag{7.4}$$

As mentioned above F and b_F are the number of frames and the number of bits in a frame. D is the width in bits and T_{ICAP} is the clock period of the ICAP interface. The reconfiguration time of the read-modify-write version of ICAP reconfiguration is simply double that of the write-only version. The data has to be processed once when read and once when written, while the bandwidth of the ICAP when reading is the same as when writing [106].

7.3.2 SRL reconfiguration

The scattering of the TLUTs in the reconfiguration memory space (and the growth of memory usage with the number of modules) can be avoided using SRLs, as shown in Figure 7.13 (b). The degrees of freedom when constructing reconfiguration paths, can be used to retain the regularity of the design in the reconfiguration paths. To achieve this goal the reconfiguration paths are chosen so that the TLUTs of one module are always coherent and the order of the different TLUTs in a module is the same. Of course there are still different ways to connect the different TLUTs. In these experiments, the ordering of the modules is the same as the ordering in the pipelined FIR filter. The ordering of the TLUTs in one module is the same as the order of appearance in the VHDL file. Further optimizations are possible.

By choosing the reconfiguration paths as mentioned above, one can take full advantage of the regularity of the design to minimize the memory usage of the specialization procedure. The pseudo-code that represents the specialization procedure in this case is shown in Figure 7.14. It is important to understand that the order in which TLUTs in a module are specialized, is the same for all modules and corresponds with the order chosen in the SRL reconfiguration paths. That is why the for loop in the pseudo code can be used in Figure 7.14. As shown in equation (7.5), the only data that is stored is the evaluation function of the PPC. Indeed, combining TLUT circuits and SRL reconfiguration results in a very memory efficient method for run-time reconfiguration of regularly structured designs.

$$M_{SRL} = PPC_{Module} \tag{7.5}$$

Using SRLs also has advantages for the reconfiguration time. Only the LUTs that have to be reconfigured are put in the reconfiguration

```
function SpecializationSRL(parameterList)
  for parameter in parameterList:
    evaluatedModule = evaluateModule(parameter);
    configureFPGA(evaluatedModule);
```

Figure 7.14: Pseudo code for the SRL specialization procedure.

paths. Increasing the number of reconfiguration paths reduces the number of bits per reconfiguration path and increases the reconfiguration speed. In the SRL case, the reconfiguration speed can thus be tuned to the application requirements.

$$T_{SRL} = \frac{M \cdot L_M \cdot 2^K \cdot T_{shift}}{R} \tag{7.6}$$

The formula for obtaining the reconfiguration time in the case of SRL reconfiguration is given above in equation (7.6). The time needed to reconfigure the design is dependent on the number of modules M, the number of TLUTs per module L_M, the number of inputs K of one SRL, the period T_{shift} of the clock frequency at which the bits are shifted in and the number of reconfiguration paths R.

7.3.3 Experimental set-up

The ideas above are illustrated by implementing a fully pipelined FIR filter, of which the coefficients are chosen as the parameters. The module in this case is an 8 by 8 bit multiplier of which one of the operands is a parameter. The parameter list $\overline{p_M}$, depicted in Figure 7.11, thus contains the specific coefficient values the FIR filter is specialized for during run-time. As discussed earlier, such a run-time reconfigurable FIR filter is 40 % more area efficient than a generic FIR filter of which the coefficients are inputs to the FPGA [19]. The memory usage of the specialization procedure is compared for ICAP and SRL reconfiguration while increasing the number of modules from 64 to 1024. Also the reconfiguration time and impact of SRL reconfiguration on the maximum clock frequency of the FIR filter are discussed.

The general characteristics of the FIR filter relevant for the equations from the previous section are $L_M = 24$, $PPC_{LUT} = 296kb$ and

Table 7.2: Number of frames (F) and shift clock period (T_{shift}) for the different FIR filters.

M	64	128	256	512	1024
F	182	339	601	1095	1422
T_{shift} (ns)	5,170	7,23	8,203	8,050	8,529
max clk (MHz)	198	190	186	156	92

$PPC_{Module} = 136kb$. To obtain the values of PPC_{LUT} en PPC_{Module} the evaluation C-functions are compiled on a Microblaze v7.10.c [104]. The FIR characteristics that are dependent on the number of modules are given in Table 7.2. This experiment is conducted on a Virtex4(xc4vlx100) [103] using ISE 10.1 software with default settings. The characteristics of the Virtex 4 ICAP relevant for the equations from the previous section are given in Table 7.3.

Table 7.3: Characteristics of a Virtex4 FPGA.

K	T_{ICAP}	D	b_F	L_F	b_A
4	10 ns	32 bit	1312 bit	80	32 bit

7.3.4 Results

7.3.4.1 Memory efficiency

In Figure 7.15 (a) the memory usage of the ICAP reconfiguration (both the write-only and the read-modify-write version) and SRL reconfiguration are shown relative to the total memory of all BRAMs of the FPGA, which is 4320 Kb [103]. It can be seen that the memory requirements of the write-only ICAP reconfiguration increase dramatically with the number of modules, consuming more than 60 % of the FPGAs BRAMs for the case with 1024 modules. Indeed, storing the constant data of the frames is very memory consuming. Even the more memory efficient read-modify-write ICAP version needs 20 % of the FPGA BRAMs to implement the FIR filter with 1024 mod-

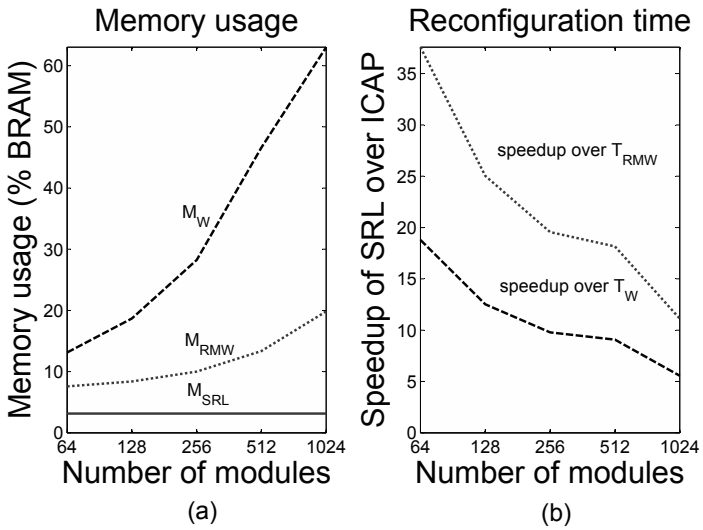


Figure 7.15: Influence of the number of modules on the memory usage (M) and reconfiguration time (T). W, RMW and SRL in the indices indicate ICAP write-only, ICAP read-modify-write and SRL reconfiguration, respectively.

ules. With SRL reconfiguration, on the other hand, even for a FIR filter with 1024 modules only 3% of the FPGA's memory is needed. As pointed out earlier, the memory usage for SRL reconfiguration is independent of the number of modules.

7.3.4.2 Reconfiguration time

In this experiment the width of the SRL reconfiguration interface is chosen the same as the ICAP interface of the Virtex4, namely 32 bits. Of course, increasing the number of reconfiguration paths would further improve the results obtained for the reconfiguration time of SRL reconfiguration.

In Figure 7.15 (b) it can be seen that, in this case, at least a 6X and at most a 37X speedup over ICAP reconfiguration can be obtained when using SRL reconfiguration. The decrease in speedup as the number of modules increases has two reasons. The number of frames increases less than linearly with the number of modules, as opposed to the total number of TLUTs $M \cdot L_M$. A second reason is the occurence of routing congestion, that reduces the clock frequency

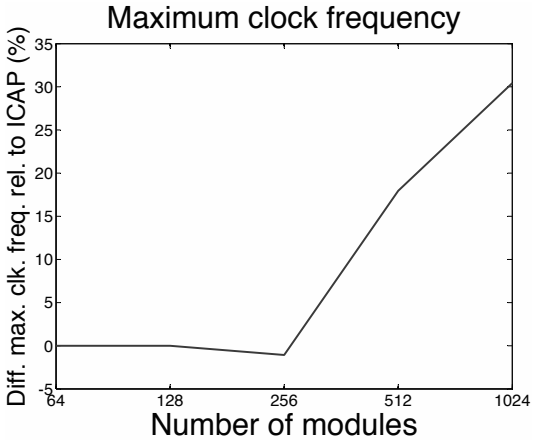


Figure 7.16: Influence of the SRL reconfiguration paths on the maximum clock frequency.

at which the bits are shifted in. These two effects can be clearly seen in Table 7.2. Note that for ICAP reconfiguration a trade-off has to be made between speed and memory-efficiency. Using SRLs results in a reconfiguration process that is both memory-efficient and fast.

In Section 7.2.2 it was shown that the reconfiguration speedup for the Virtex 2 Pro was around two orders of magnitude. The difference with the Virtex 4 is mainly due to the improved ICAP interface. The ICAP of the Virtex4 has a port width of 32 bits and runs at 100 MHz, while that of the Virtex 2 Pro has a width of 8 bits and runs at 66 MHz. The configuration memory of the Virtex 4 also consists of frames that contain less configuration bits.

7.3.4.3 Impact on maximum clock frequency

The maximum clock frequency of the designs without SRLs can be found in the last row of Table 7.2. The decrease in maximum clock frequency as the number of modules increases is due to routing congestion. The FIR filter with 64 modules occupies 4% of the slices of the FPGA, while the one with 1024 modules takes 87%. Figure 7.16 presents the difference between the maximum clock frequency of the design with SRL reconfiguration paths and the design without, which is reconfigured using ICAP reconfiguration. It can be clearly seen that the negative impact of SRL reconfiguration on the

maximum clock frequency is very limited. The decrease is never bigger than 1%. In fact, as the number of modules increases, it could be expected that due to increased routing congestion the maximum clock frequency to be worse when the SRL reconfiguration paths are present. The maximum clock frequency of the FIR filter however improves compared to ICAP reconfiguration. The presence of the SRL reconfiguration paths of course influences the packing and placement and routing process and in the case of the FIR filter this seems to be a positive influence. As discussed in Section 7.2.2, this effect was also observed in [45].

Chapter 8

Conclusions and Future Work

Dynamic Partial Configuration (DPR) makes it possible to implement different circuits on the same FPGA area, called the *reconfigurable region*. Since several mutually exclusive functions of an application can share the same area, a more area efficient implementation is obtained. This possibly results in the use of a smaller and thus cheaper FPGA. However, compared to a static FPGA implementation, a relatively long time, called the reconfiguration time, is now needed to rewrite the configuration memory of the reconfigurable region when a different function is needed. In the experiments of this work, the reconfiguration time for the conventional DPR flow was estimated to be around 1.5 ms on average. In general, the overhead associated with the reconfiguration process is called the *reconfiguration overhead*. This can consist, among others, of the hardware resources needed and the energy consumed during reconfiguration.

Two examples were given in this dissertation of applications where the reconfiguration time constraints are so strict that they possibly hinder the efficient use of DPR. A first example is the network intrusion detection system described in [76], where reconfiguration has to happen fast enough so that no network packets are dropped. A second example is an imaging application where the reconfiguration possibly happens per image frame [10]. In the case of [76] there are only 2 contexts, corresponding to different networking protocols. The work in [10] considers a video chain with a limited number of processing steps. These are therefore both also good examples of limited-context applications.

This concluding chapter, just as the dissertation itself, is divided in two parts. The first part concludes on the research on limited-context DPR, discussed in Chapter 4 and Chapter 5. The second part concludes on the research using shift-register LUTs (SRLs), which was discussed in Chapter 6 and Chapter 7. In the PhD dissertation of dr. ir. Karel Bruneel, Dynamic Circuit Specialization (DCS) techniques are presented [17]. As discussed in Chapter 3, DCS allows for an efficient DPR implementation of systems with slowly varying signals called parameters. In these systems there is one design that is specialized for certain parameter signals during run-time.

The first part of this research considers a very different type of reconfiguration than in [17], namely one where several (and not only one) larger functions, that are used mutually exclusive in time, are time-multiplexed on the same FPGA area called the reconfigurable region. In this first part of the dissertation it is assumed all Look-up Tables (LUTs) are reconfigured, together with the dynamic portions of the routing (expressed in bits in Chapter 4 and in frames in Chapter 5).

The second part builds further on the TMAP tool flow that was presented in [17]. In this tool flow only the LUT bits are reconfigured. The second part of this dissertation improves the TMAP tool flow by introducing SRL reconfiguration. So, in the second part of this dissertation only the LUTs are reconfigured during run-time while the routing remains fixed.

8.1 Limited-context Dynamic Partial reconfiguration

The conventional DPR tool flow implements the different circuits in the reconfigurable region separately and does not optimize the reconfiguration overhead. The number of circuits implemented in the reconfigurable region is only limited by the amount of memory available to store the configuration bit streams needed for reconfiguration. In some cases, however, the number of circuits implemented in the reconfiguration region is limited, a situation called *limited-context DPR*.

8.1.1 Conclusions

In the first part of this dissertation, I consider an approach for limited-context DPR where the different circuits are not implemented separately as in the conventional DPR flow. Instead, from a certain point in the flow on, a combined implementation and optimization takes place that also takes into consideration the reconfiguration overhead. This approach is only viable when the number of circuits is limited, as is the case in limited-context DPR. The configuration memory of current commercial FPGAs consist mostly of bits that control the state of the configurable interconnection network. The approach taken in this dissertation is therefore to attempt to reduce the reconfiguration overhead of the interconnections.

Exploring novel tool flows for limited-context dynamic partial reconfiguration The combined implementation approach can take place starting at different points in the tool flow. In Chapter 4, I explored three different tool flows that start the combined implementation at a different point in the flow: before placement, during placement and during routing. The first tool flow I researched combined the different circuits before placement using a previously developed technique found in literature called *edge matching*. In this tool flow, the placer and router from the Dynamic Circuit Specialisation tool flow, called TPlace and TRoute, were used for further implementation in the reconfigurable region. The second tool flow combines the circuits during placement and optimizes the *total wire length* of all the circuits using the same cost function as TPlace. The third and final tool flow optimizes the *individual wire lengths* of the circuits during placement and only performs combined routing using TRoute.

A first conclusion of this exploration was that edge matching increases the wire length of the implemented circuits excessively, sometimes up to 192% for two circuits, without contributing to the reduction in reconfiguration overhead. For the three different tool flows considered, a reduction in reconfiguration overhead, expressed as number of configuration bits, of around 35% was achieved. It turns out that most of the reduction in reconfiguration overhead is generated through the combined routing using TRoute.

A comparison of the two wire length driven tool flows results in the finding that the wire length of the circuits is best optimized individually during placement. When implementing two circuits,

the individual wire length optimization approach increased the wire length of the circuits on average with 11%, compared to 26% for the total wire length optimization approach.

A tool flow for limited-context DPR therefore should reuse the steps of the conventional DPR flow up until placement and only perform a combined routing, as is the case in the third tool flow considered. For this tool flow, experiments were also done with more than two circuits. The tool flow is able to find a solution, when the number of circuits is not more than 4. The reconfiguration overhead only decreases with a few percent when the number of circuits increases and thus stays above 30%. The wire length increase associated with this reduction in reconfiguration overhead is around 18% and 24% for three and four circuits, respectively.

Reducing the overhead of limited-context DPR in a frame-based reconfiguration approach In the exploration of novel tool flows for limited-context DPR, the reconfiguration overhead is expressed as the number of bits that needs to be rewritten in the configuration memory. However, the configuration memory of current commercial FPGAs is organized in frames. A *configuration frame* needs to be rewritten completely, even when it contains only one dynamic bit. Although many lessons were learned from the techniques developed in the previous chapter, these would only be useful if the routing's configuration memory would be bit-addressable. Therefore, in Chapter 5 other tool flows are developed that can be used directly in a frame-based reconfiguration approach. These techniques were developed based on lessons learned from the exploration discussed above. Chapter 5 presented two flows: the StaticRoute and the ClusterRoute tool flow.

In the *StaticRoute* flow the routing happens in two steps. In a first step the configuration memory of the reconfigurable region's routing switches is divided into a static and a dynamic part. Then, in a second step the interconnections of all circuits are routed simultaneously using StaticRoute in such a way that the dynamic bits are clustered in the dynamic part.

Experiments done in the academic VTR framework suggest that, using the StaticRoute tool flow, the reconfiguration overhead can be reduced by around 34%. The tool flow only finds a solution when the number of circuits is limited to at most four. Of course, the reduction in reconfiguration overhead does not come for free. Because of the

Table 8.1: Overview of the reduction in clock frequency (RCF) or wire length increase (WLI) and reduction in reconfiguration overhead for the TRoute, StaticRoute (SR) and ClusterRoute (CR) algorithm, for increasing number of circuits (all in % relative to conventional DPR).

	RCF/WLI			RRO		
Nr. of circuits	TR	SR	CR	TR	SR	CR
2	11	5	5	36	34	32
3	18	8	11	35	34	27
4	22	14	21	33	34	19

(in %)

combined routing approach, it is associated with a reduction in maximum operating clock frequency. This reduction in maximum clock frequency is limited on average to around 6% for two circuits and increases to around 9% and 15% for three and four circuits respectively. When we assume a fixed clock frequency of the circuits during run-time, as is common in real-life applications, these reductions improve to 5% for two circuits and around 8% and 14% for three and four circuits respectively. The average values of these reductions in maximum clock frequency (RCF) and the reductions in reconfiguration overhead (RRO) are summarized in Table 8.1 for both the ClusterRoute and StaticRoute algorithm. The wire length increase (WLI) results and RRO results of the individual wire length approach (in bits instead of frames) using TRoute from Chapter 4 are added for comparison.

Using the StaticRoute tool flow is also associated with a slowdown of off-line compilation time of the tool flow. This slowdown is mostly because this flow also takes the reconfiguration overhead into consideration and thus solves a much more complex problem. Compared to the conventional DPR flow a slowdown occurs of 7$\times$ for two circuits. This slowdown increases to 8$\times$ and 10$\times$ for three and four circuits, respectively.

Finally, an automatic combined DPR flow, called the *ClusterRoute tool flow*, is presented in which it is not necessary to manually make

a distinction between static and dynamic configuration frames before routing. The ClusterRoute tool flow automatically clusters the dynamic bits in fewer configuration frames. It achieves a reduction of reconfiguration overhead of 32% for two circuits. This reduction decreases to 27% and 19% for three and four circuits respectively. As is the case for the StaticRoute flow, the reduction in reconfiguration overhead in the ClusterRoute flow is also associated with a reduction in maximum clock frequency. This reduction is around 5% for two circuits and increases to around 11% and 21%, for three and four circuits respectively. The reduction in maximum clock frequency of the ClusterRoute flow is thus somewhat bigger than the reduction obtained in the StaticRoute flow. ClusterRoute, on the other hand, performs the implementation completely automatic, as opposed to the StaticRoute flow, where some of the configuration frames first have to be marked static.

The impact of the reduction in maximum clock frequency on the overall performance of the application will depend on the type of application. If the application needs to run at the absolute highest clock frequency, these combined implementation techniques will not be applicable. However, this is not always the case. It could be that the constraints that are imposed on the system are not that stringent. FPGAs often rely more on massive parallelism than on high clock frequencies. There are also applications that are more restricted by the bandwidth offered by FPGAs. Finally, clocks on FPGAs are mostly derived by dividing or multiplying a round base clock (of for example 100 MHz) using clock generators. It is not unusual that a circuit with a maximum clock frequency of for example 217 MHz is run at a round 200MHz.

The ClusterRoute flow is also associated with a considerable slowdown of the off-line compilation time. Compared to the conventional DPR flow a slowdown of 14$\times$ occurs for two circuits. This slowdown increases to 18$\times$ and 23$\times$, for three and four circuits respectively. The slowdown of the ClusterRoute flow is greater than the StaticRoute flow's slowdown. This reason for this is that the problem that ClusterRoute solves is more complex. In the StaticRoute flow the problem is broken down in two steps, which reduces the complexity .

When reconfiguration only happens once-in-a-while the approaches that use DPR will have a higher functional density than the static approach since they require much less area and all of the

hardware resources are used at all time. The conventional DPR approach is most efficient since the clock frequency at which the circuits operate is maximal and the area minimal, while the reconfiguration time plays less of a role as the reconfiguration rate is very low. When the configuration rate corresponds to a circuit re-use of around 500 thousand clock cycles, the StaticRoute approach will have a better functional density than the conventional DPR approach. The StaticRoute algorithm always outperforms the ClusterRoute algorithm what functional density is concerned. Note however, that the ClusterRoute algorithm is more automated. If an application is constrained in area and the use of DPR is necessary, the StaticRoute approach clearly outperforms conventional DPR.

8.1.2 Future Work

As discussed in the previous section, a combined implementation approach that also takes into account the reconfiguration overhead is associated with a considerable slowdown of the compilation. A first possible track for future work could explore ways to improve the compilation time of the tool flow. The current version of the tool flow could be analysed and optimized for memory usage and compilation time. The algorithm can also possibly be accelerated by trading off some memory usage for a reduced compilation time. After the compilation time is shortened, it can be researched if the cost function of StaticRoute and ClusterRoute can be further improved. Also, the relation between the number of routing iterations of the algorithm and the number of circuits that can be implemented can be researched more thoroughly after the compilation time is reduced.

As discussed in Chapter 5, the way the configuration memory is built up is considered proprietary information and is not disclosed by FPGA manufacturers. There clearly is a need for an academic framework, similar to the VTR framework for the FPGA fabric, in which experiments can be done concerning the configuration memory.

Using such an academic framework, a different opportunity for future work is the design of a configuration memory that is more tailored to DPR. The trade-off between chip area and granularity of the configuration memory could be, for example, researched. A smaller granularity increases the efficiency of the reconfiguration process, but also makes the configuration memory bigger and more complex.

An interesting granularity would be one that has the same width as the configuration port. Of course, it would also be interesting to look at how configuration bits are grouped in frames in current commercial FPGAs and if this can be improved, taking the reconfiguration overhead of DPR into consideration. Finally, it could also be researched what the trade-off is between the area cost and configuration speed of a frame.

A hybrid configuration memory could also be researched in which the configuration frames are split up in two groups: the normal frames and frames optimized for configuration speed. The configuration would then also be split up in two phases, each one configuring a specific type of frame. A new tool flow can then be developed, such that during run-time mostly the faster frames are reconfigured.

8.2 SRL reconfiguration of regularly structured designs

In certain circuits a subset of input signals, called parameters, only vary infrequently. The Dynamic Circuit Specialization method, implements these designs efficiently by generating a configuration offline that does not only contain constant 0's and 1's but also bits that are a Boolean function of the parameter signals. This type of configuration is called a parameterized configuration and the Boolean functions are called parameterized bits. Whenever the parameters change value, the configuration manager only needs to evaluate the parameterized bits and reconfigure the FPGA. The *TLUT tool flow* is able to automatically generate such parameterized configurations in which only the truth table bits are parameterized.

In a Shift Register LUT, the truth table configuration bits are also arranged as a shift register of which the input and the output are accessible from the configurable routing. Therefore these configuration bits are not only accessible through the FPGA's configuration ports (e.g. the Internal Configuration Access Port or ICAP), but also through the shift inputs of the SRLs. SRLs can thus be used to reconfigure the truth table contents of a LUT during run-time, hence the term *SRL reconfiguration*.

8.2.1 Conclusions

Integration of SRL reconfiguration in the TLUT tool flow One of the findings of my research is that making appropriate use of SRLs for reconfiguration requires adding timing constraints to make the tools treat the SRLs as conventional LUTs and not as shift registers. Also a proof-of-concept design of a hardware block, called HWSRL, is presented that interfaces between the configuration manager and the reconfigurable hardware. Finally, in order to make the truth table bits of multiple LUTs accessible from the configuration manager, they are grouped and each group is arranged as a larger shift register, called a reconfiguration path, by connecting the shift out of an SRL to the shift in of the next SRL.

Exploring the benefits and overhead of SRL reconfiguration for regularly structured designs In regularly structured designs, the same hardware module is instantiated many times. In the experiments of Chapter 7 the *reconfiguration speed-up* of SRL reconfiguration compared to the convention Internal Configuration Access Port (ICAP) is quantified for regularly structured designs. On the Virtex 2 Pro, a speed-up of around 2 orders of magnitude is obtained, while on the Virtex 4 the speed-up was between 6× and 37×. For both architectures the speed-up is very significant. This is because SRLs are more tailored to LUT-only reconfiguration. As explained before, the reconfiguration granularity of the ICAP is a frame. It thus also reconfigures LUTs that do not need to be changed during run-time. The differences in reconfiguration speed-up between the architectures are explained by the difference between the ICAP implementation on each. On the Virtex 2 Pro the width of the configuration port is 8 bit and reconfiguration happens at a clock frequency of 66 MHz. For the Virtex 4 this is 32 bit and 100 MHz. Also, the frames on the Virtex 4 contain fewer configuration bits.

When reconfiguring regularly structured designs, it is also shown that using SRL reconfiguration, the *memory usage* of the configuration manager can be kept constant, regardless of the number of modules. Experiments with an adaptive filtering application on a medium-sized Virtex 4 FPGA show that the memory requirements of write-only ICAP reconfiguration increase dramatically with the number of modules, consuming more than 60% of the FPGA's BRAMs for the case with 1024 modules. Even the more memory efficient read-

modify-write ICAP version needs 20% of the FPGA BRAMs to implement the adaptive filter with 1024 modules. With SRL reconfiguration, on the other hand, even for an adaptive filter with 1024 modules only 3% of the FPGA's memory is needed. As the write-only version of ICAP reconfiguration is faster than the read-modify-write ICAP version, a trade-off needs to be made between reconfiguration speed and memory efficiency. The use of SRLs, on the other hand, results in a reconfiguration process that is both memory-efficient and fast.

Because the SRL reconfiguration paths also make use of the configurable interconnection network, it could be that they interfere with the *timing of the design*. Reconfiguration paths that follow the structure of the design result in a smaller reduction in maximum clock frequency than randomly chosen reconfiguration paths. In the experiments in Chapter 7 it was shown that using reconfiguration paths following the regular structure, the clock frequency of the design reduces somewhat, but that this reduction is mostly limited. For both the experiments done on the Virtex 2 Pro and Virtex 4, the decrease in maximum clock frequency stays within 10%. These results are similar to the result found in [45]. In the experiments done in Chapter 7, it was also observed that the number of reconfiguration paths seems to have little influence on the clock frequency. The largest effect comes from the number of modules of the design. This could indicate that more routing congestion arises as the size of the design increases.

8.2.2 Future Work

In my research I have shown that the use of SRLs offers many benefits over ICAP reconfiguration when only LUTs need reconfiguration. They improve the speed of the reconfiguration process and the memory efficiency. It would also be interesting to research how SRLs affect the energy consumption of the reconfiguration process.

As explained in Chapter 6, a Configurable Logic Block contains dedicated routing between adjacent SRLs. To connect these SRLs therefore no wires from the configurable interconnection network are needed. Another possible direction for future research is to explore other architectural changes from which SRL reconfiguration could benefit.

In this dissertation, the reconfiguration paths are chosen based on the regular structure of the design under consideration. LUTs from a

same module were placed together in a reconfiguration path. The order of LUTs from a module was chosen in the order of their appearance in the VHDL file. In [45], a general method is presented that solves a multiple travelling salesman problem based on the placement of the design to determine how the SRLs are placed in reconfiguration paths. It would be interesting to see if this method could be specialised for regularly structured designs. In these designs, it is fixed how SRLs are grouped in SRL reconfiguration paths. The order of modules and the order of SRLs in one module are the only remaining degrees of freedom.

Bibliography

[1] Bleeding edge threats website.

[2] A. Ahmad, A. Amira, P. Nicholl, and B. Krill. Fpga-based ip cores implementation for face recognition using dynamic partial reconfiguration. *Journal of real-time image processing*, 8(3):327–340, 2013.

[3] B. Al Farisi. Herconfiguratie van LUTs via hun schuifregisterfunctionaliteit. Master's thesis, 2009.

[4] B. Al Farisi, K. Bruneel, J. M. P. Cardoso, and D. Stroobandt. An automatic tool flow for the combined implementation of multi-mode circuits. In *Proceedings - Design, Automation, and Test in Europe Conference and Exhibition*, pages 821–826, Grenoble, France, 2013.

[5] B. Al Farisi, E. Vansteenkiste, K. Bruneel, and D. Stroobandt. A novel tool flow for increased routing configuration similarity in multi-mode circuits. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI 2013 (ISVLSI13)*, pages 96–101, Natal, Brazil, 2013.

[6] M. Al Kadi, P. Rudolph, D. Gohringer, and M. Hubner. Dynamic and partial reconfiguration of zynq 7000 under linux. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013.

[7] Altera. *Configuration, Design Security, and Remote System Upgrades in Stratix IV Devices*, September 2012.

[8] Altera. *Engineering Change Management with the Chip Planner*, 2012.

[9] Altera. *Stratix IV Device Handbook*, January 2014.

[10] J. Basteleus. Dynamische circuitspecialisatie voor autostereo 3d tv. Master's thesis, 2014.

[11] A. Becher, F. Bauer, D. Ziener, and J. Teich. Energy-aware sql query acceleration through fpga-based dynamic partial reconfiguration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[12] T. Becker, M. Koester, and W. Luk. Automated placement of reconfigurable regions for relocatable modules. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3341–3344. IEEE, 2010.

[13] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[14] A. Biere. The AIGER and-inverter graph (AIG) format. *Available at fmv. jku. at/aiger*, 2007.

[15] A. M. R. Brayton. Scalable logic synthesis using a simple circuit structure.

[16] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

[17] K. Bruneel. *Efficient Circuit Specialization for Dynamic Reconfiguration of FPGAs*. PhD thesis, Ghent University, 2011.

[18] K. Bruneel, W. Heirman, and D. Stroobandt. Dynamic data folding with parameterizable FPGA configurations. *ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS*, 16(4):29, 2011.

[19] K. Bruneel, F. Mostafa Mohamed Ahmed Abouelella, and D. Stroobandt. Automatically mapping applications to a self-reconfiguring platform. In *Proceedings of DATE*, pages 964–969, 2009.

[20] K. Bruneel and D. Stroobandt. Automatic generation of run-time parameterizable configurations. In U. Kebschull,

M. Platzner, and T. J., editors, *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 361–366, Heidelberg, 9 2008. Kirchhoff Institute for Physics.

[21] K. Bruneel and D. Stroobandt. TRoute : a reconfigurability-aware FPGA router. In *LECTURE NOTES IN COMPUTER SCIENCE*, volume 5992, pages 207–218, Berlin, Germany, 2010. Springer Verlag Berlin.

[22] E. Cetin, O. Diessel, L. Gong, and V. Lai. Towards bounded error recovery time in fpga-based tmr circuits using dynamic partial reconfiguration. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4. IEEE, 2013.

[23] C. Chavet, C. Andriamisaina, P. Coussy, E. Casseau, E. Juin, P. Urard, and E. Martin. A design flow dedicated to multi-mode architectures for DSP applications. In *IEEE/ACM International Conference on Computer-Aided Design, 2007*, pages 604–611. IEEE, 2007.

[24] D. Chen and J. Cong. Daomap: A depth-optimal area optimization mapping algorithm for FPGA designs. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 752–759. IEEE Computer Society, 2004.

[25] W. Chen, Y. Wang, X. Wang, and C. Peng. A new placement approach to minimizing FPGA reconfiguration data. In *International Conference on Embedded Software and Systems ICESS'08.*, pages 169–174. IEEE, 2008.

[26] C. Claus, R. Ahmed, F. Altenried, and W. Stechele. Towards rapid dynamic partial reconfiguration in video-based driver assistance systems. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 55–67. Springer, 2010.

[27] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[28] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, 1994.

[29] J. Cong, C. Wu, and Y. Ding. Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 29–35. ACM, 1999.

[30] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto. Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(5):662–675, 2009.

[31] A. Corporation. *Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs*. Altera Corporation, 2010.

[32] A. Corporation. *Introducing Innovations at 28 nm to Move Beyond Moores Law*. Altera Corporation, 2010.

[33] P. Coussy, G. Lhairech-Lebreton, D. Heller, and E. Martin. Gaut–a free and open source high-level synthesis tool. *IEEE Design Automation and Test in Europe–University Booth*, 2010.

[34] A. M. Dehon. *Reconfigurable architectures for general-purpose computing*. PhD thesis, Massachusetts Institute of Technology, 1996.

[35] M. Della Torre, U. Malik, and O. Diessel. A configuration system architecture supporting bit-stream compression for fpgas. In *Advances in Computer Systems Architecture*, pages 415–428. Springer, 2005.

[36] C. Dennl, D. Ziener, and J. Teich. Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 25–28. IEEE, 2013.

[37] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. In *IEE Proceedings Computers and Digital Techniques*, volume 147, pages 181–188. IET, 2000.

[38] J. Divyasree, H. Rajashekar, and K. Varghese. Dynamically reconfigurable regular expression matching architecture. In *ASAP '08*, 2008.

[39] F. Duhem, F. Muller, and P. Lorenzini. Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 253–260. Springer, 2011.

[40] E. Eto. Difference-based partial reconfiguration, 2003.

[41] S. G. Hansen, D. Koch, and J. Torresen. High speed partial run-time reconfiguration using enhanced ICAP hard macro. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 174–180. IEEE, 2011.

[42] M. Hariyama, W. H. Muthumala, and M. Kameyama. Dynamically reconfigurable gate array based on fine-grained switch elements and its cad environment. In *Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian*, pages 155–158. IEEE, 2006.

[43] HES Group, Ghent University. TMAP toolflow. https://github.com/UGent-HES/tlut_flow, 2012.

[44] K. Heyse. Automatiseren van srl-herconfiguratie. Master's thesis, 2011.

[45] K. Heyse, B. Al Farisi, K. Bruneel, and D. Stroobandt. Automating reconfiguration chain generation for SRL-based run-time reconfiguration. In *Lecture notes in computer science*, volume 7199, pages 1–12, Berlin, Germany, 2012. Springer.

[46] K. Heyse, K. Bruneel, and D. Stroobandt. Mapping logic to reconfigurable FPGA routing. In *22nd International Conference on Field Programmable Logic and Applications, Proceedings*, pages 315–321, Piscataway, NJ, USA, 2012. IEEE.

[47] M. Hubner, D. Gohringer, J. Noguera, and J. Becker. Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[48] C. Huriaux, O. Sentieys, and R. Tessier. Fpga architecture support for heterogeneous, relocatable partial bitstreams. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[49] X. Inc. *Partial Reconfiguration Of Virtex FPGAs in ISE 12*. Xilinx Inc., 2010.

[50] X. Inc. *Partial Reconfiguration User Guide*. Xilinx Inc., 2010.

[51] X. Inc. *WP374: Partial Reconfiguration Of Xilinx FPGAs Using ISE Design Suite*. Xilinx Inc., 2012.

[52] X. Inc. *UG702: Partial Reconfiguration User Guide*. Xilinx Inc., 2013.

[53] X. Inc. *DS190: Zynq-7000 All Programmable SoC Overview*. Xilinx Inc., 2014.

[54] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Proceedings. 19th IEEE International Symposium on Parallel and Distributed Processing*, pages 151b–151b. IEEE, 2005.

[55] I. O. Kennedy. A dynamically reconfigured umts multichannel complex code matched filter. In *IEEE International Conference on Field-Programmable Technology, 2005*, pages 199–206. IEEE, 2005.

[56] I. O. Kennedy. Implementation of low frequency finite state machines using the virtex SRL16 primitive. In *Proceedings of FPL*, 2007.

[57] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[58] V. Kizheppatt and S. Fahmy. Zycap: Efficient partial reconfiguration management on the xilinx zynq. 2014.

[59] D. Koch, C. Beckhoff, and J. Teich. Minimizing internal fragmentation by fine-grained two-dimensional module placement for runtime reconfiguralble systems. In *17th IEEE Symposium on Field Programmable Custom Computing Machines, FCCM'09.*, pages 251–254. IEEE, 2009.

[60] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Rapidsmith: Do-it-yourself cad tools for xilinx fpgas. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 349–355. IEEE, 2011.

[61] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and single-driver wires in FPGA interconnect. In *IEEE International Conference on Field-Programmable Technology, 2004*, pages 41–48. IEEE, 2004.

[62] J. Leonard and W. H. Mangione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. In *Proc. International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 151–160, 1997.

[63] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, et al. The stratix ii logic and routing architecture. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20. ACM, 2005.

[64] Z. Li and S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195. ACM, 2002.

[65] J. V. Lindholm, I. L. McEwen, and J. T. Young. Routing with frame awareness to minimize device programming time and test cost, Dec. 12 2006. US Patent 7,149,997.

[66] J. Luu and J. Rose. Vpr 6.0 user manual, January 2012.

[67] P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M. Di Ciano, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, et al. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP Journal on Embedded Systems*, 2008:1, 2008.

[68] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. Heuristics for area minimization in lut-based FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(11):2331–2340, 2006.

[69] T. Marconi, J. Y. Hur, K. Bertels, and G. Gaydadjiev. A novel configuration circuit architecture to speedup reconfiguration and relocation for partially reconfigurable devices. In *IEEE 8th Symposium on Application Specific Processors (SASP), 2010*, pages 87–92. IEEE, 2010.

[70] M. Masud and S. Wilton. A new switch block for segmented fpgas. In P. Lysaght, J. Irvine, and R. Hartenstein, editors, *Field Programmable Logic and Applications*, volume 1673 of *Lecture Notes in Computer Science*, pages 274–281. Springer Berlin Heidelberg, 1999.

[71] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117. ACM, 1995.

[72] P. B. Minev and V. S. Kukenska. The virtex-5 routing and logic architecture. *Annual Journal of Electronics, Technical University of Sofia*, 3:107–110, 2009.

[73] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. Combinational and sequential mapping with priority cuts. In *IEEE/ACM International Conference on Computer-Aided Design, 2007 (ICCAD 2007)*, pages 354–361. IEEE, 2007.

[74] F. Nava, D. Sciuto, M. D. Santambrogio, S. Herbrechtsmeier, M. Porrmann, U. Witkowski, and U. Rueckert. Applying dynamic reconfiguration in the mobile robotics domain: A case study on computer vision algorithms. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):29:1–29:22, Aug. 2011.

[75] K. Papadimitriou, A. Dollas, and S. Hauck. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems*, 4(4):36:1–36:24, Dec. 2011.

[76] S. Pontarelli, C. Greco, E. Nobile, S. Teofili, and G. Bianchi. Exploiting dynamic reconfiguration for fpga based network intrusion detection systems. In *2010 International Conference on Field Programmable Logic and Applications (FPL)*, pages 10–14. IEEE, 2010.

[77] K. Prasad Raghuraman, H. Wang, and S. Tragoudas. A novel approach to minimizing reconfiguration cost for LUT-based FPGAs. In *18th International Conference on VLSI Design*, pages 673–676. IEEE, 2005.

[78] K. Raghuraman, H. Wang, and S. Tragoudas. Minimizing FPGA reconfiguration data at logic level. In *Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 219–224. IEEE Computer Society, 2006.

[79] J. C. Rodriguez and K. F. Ackermann. Leveraging partial dynamic reconfiguration on zynq soc fpgas. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–6. IEEE, 2014.

[80] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR project: architecture and CAD for FPGAs from verilog to routing. In *Proceedings of FPGA*, pages 77–86. ACM, 2012.

[81] B. Rousseau, P. Manet, T. Delavallée, I. Loiselle, and J.-D. Legat. Dynamically reconfigurable architectures for software-defined radio in professional electronic applications. In *Design Technology for Heterogeneous Embedded Systems*, pages 437–455. Springer, 2012.

[82] M. Rullmann and R. Merker. Maximum edge matching for reconfigurable computing. In *20th International Parallel and Distributed Processing Symposium, 2006 (IPDPS 2006)*, page 8. IEEE, 2006.

[83] T. Sasao. An application of 16-valued logic to design of reconfigurable logic arrays. In *ISMVL*, page 40, 2007.

[84] T. Sasao and H. Nakahara. Implementations of reconfigurable logic arrays on FPGAs. In *International Conference on Field-Programmable Technology*, 2007.

[85] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. *Computers and Digital Techniques*, 153(3):157 – 164, 2006.

[86] L. Shang and N. K. Jha. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 345. IEEE Computer Society, 2002.

[87] S. Singh, J. Hogg, and D. McAuley. Expressing dynamic reconfiguration by partial evaluation. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1996.

[88] A. M. Smith, G. A. Constantinides, and P. Y. Cheung. Area estimation and optimisation of FPGA routing fabrics. In *International Conference on Field Programmable Logic and Applications FPL 2009.*, pages 256–261. IEEE, 2009.

[89] I. Sourdis, J. Bispo, J. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51:99–121, 2008. 10.1007/s11265-007-0131-0.

[90] H. Tan and R. F. DeMara. A physical resource management approach to minimizing FPGA partial reconfiguration overhead. In *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pages 1–5. IEEE, 2006.

[91] S. Trimberger. Scheduling designs into a time-multiplexed fpga. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 153–160. ACM, 1998.

[92] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 22–28. IEEE, 1997.

[93] E. Vansteenkiste, B. Al Farisi, K. Bruneel, and D. Stroobandt. TPaR : place and route tools for the dynamic reconfiguration of the fpga's interconnect network. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 33(3):370–383, 2014.

[94] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. A connection router for the dynamic reconfiguration of fpgas. In *Lecture*

Notes in Computer Science, volume 7199, pages 357–364, Berlin, Germany, 2012. Springer.

[95] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. Maximizing the reuse of routing resources in a reconfiguration-aware connection router. In *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 322–329, Oslo, Norway, 2012. IEEE.

[96] K. Vipin and S. A. Fahmy. A high speed open source controller for FPGA partial reconfiguration. In *International Conference on Field-Programmable Technology (FPT)*, pages 61–66. IEEE, 2012.

[97] Y. Wakaba, S. Wakabayashi, S. Nagayama, and M. Inagi. An area efficient regular expression matching engine using partial reconfiguration for quick pattern updating. *IPSJ Transactions on System LSI Design Methodology*, 7(0):110–118, 2014.

[98] M. Wirthlin and B. Hutchings. Improving functional density using run-time circuit reconfiguration FPGAs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(2):247–256, June 1998.

[99] M. J. Wirthlin. Constant coefficient multiplication using look-up tables. *J. VLSI Signal Process. Syst.*, 36(1):7–15, 2004.

[100] Xilinx. *Dynamic constant coefficient multiplier v2.0*.

[101] Xilinx. *XAPP256: FIFOs Using Virtex-II Shift Registers*. Xilinx, 2005.

[102] Xilinx. *UG012: Virtex-2 Pro and Virtex-2 Pro X FPGA User Guide*. Xilinx, 2007.

[103] Xilinx. *UG070: Virtex-4 FPGA User Guide*. Xilinx, 2008.

[104] Xilinx. *UG081: Microblaze processor reference guide(v9.2)*. Xilinx, 2008.

[105] Xilinx. *UG071: Virtex-4 FPGA Configuration User Guide*. Xilinx, 2009.

[106] Xilinx. *DS586: LogiCORE IP XPS HWICAP*. Xilinx, 2010.

[107] Xilinx. *UG190: Virtex-5 FPGA User Guide*. Xilinx, 2012.

[108] Xilinx. *UG364: Virtex-6 FPGA Configurable Logic Block*. Xilinx, 2012.

[109] Xilinx. *UG474: 7 Series FPGAs Configurable Logic Block*. Xilinx, 2014.

[110] Xilinx. *UG574: UltraScale Architecture Configurable Logic Block*. Xilinx, 2015.

[111] Xilinx, Inc. *MicroBlaze Processor Reference Guide, UG081, v14.1*, 2012.

[112] S. Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer, 1991.